Why do we want to study Security?

- It is important
- There is never a dull day!
- It is fun!
Security is Important

- An increasing part of our business, social, and personal life involves internet-connected computer systems
  - Web, email, social networks, entertainment, . . .
  - Mobile computing
  - Cyber-physical systems
  - Internet of things
- Protecting the security and privacy of our digital interactions is critical
  - Most of them involve networked systems and applications
There is never a dull day!

- Every day, we hear news of yet another high-profile hack, data theft, etc.
- New vulnerabilities surface all the time, and we have to find new solutions
- High-stakes game where attackers and defenders innovate constantly in order to stay ahead of each other
NonPetya ransomware forced Maersk to reinstall 4000 servers, 45000 PCs

Moller-Maersk puts cost of cyber attack at up to $300m

A Mysterious Hacker Group Is On a Supply Chain Hijacking

A group of likely Chinese hackers has poisoned the software of at least six companies in just the past three years.

WhatsApp discovers 'targeted' surveillance attack
Hackers ground 1,400 passengers at Warsaw in attack on airline's computers

Polish state-owned airline LOT suffers hacking assault on ground systems that causes 10 national and international flights to be cancelled
An Unprecedented Look at Stuxnet, the World’s First Digital Weapon

BY KIM ZETTER 11.03.14 | 6:30 AM | PERMALINK
How A Coffee Machine Infected Factory Computers with Ransomware

By Waqas on July 28, 2017

It’s no surprise that the Internet of Things (IoT) devices are highly vulnerable to cyber attacks but who would know a time would come when these devices will become a security threat to institutions?

A few months ago researchers exposed life threatening vulnerabilities in IIoT (Industrial Internet of Things) devices specifically Industrial robots. In their findings, robots could be hacked, but in this case, we are about to discuss a smart coffee machine or an Internet connected coffee machine.

More: San Francisco Railway' Fare System Hacked for 100 Bitcoin Ransom
Armed With Facebook 'Likes' Alone, Researchers Can Tell Your Race, Gender, and Sexual Orientation

REBECCA J. ROSEN | MAR 12 2013, 2:59 PM ET

But the deeper aspects of your personality remain hard to detect.
“Unauthorized code” in Juniper firewalls decrypts encrypted VPN traffic

Backdoor in NetScreen firewalls gives attackers admin access, VPN decrypt ability.

by Dan Goodin - Dec 17, 2015 6:50pm EST

An operating system used to manage firewalls sold by Juniper Networks contains unauthorized code that surreptitiously decrypts traffic sent through virtual private networks, officials from the company warned Thursday.

It's not clear how the code got there or how long it has been there. An advisory published by the company said that NetScreen firewalls using ScreenOS 6.2.0r15 through 6.2.0r18 and 6.3.0r12 through 6.3.0r20 are affected and require immediate patching. Release notes published by Juniper suggest the earliest vulnerable versions date back to at least 2012 and possibly earlier. There's no evidence right now that the backdoor was put in other Juniper OSes or devices.

“During a recent internal code review, Juniper discovered unauthorized code in ScreenOS that could allow unauthorized third parties to intercept and decrypt VPN encrypted traffic,” the company said in a statement. It added that it was unaware of any exploitation of the backdoor or its use to conduct unauthorized activity.

The advisory cautions users to apply the patches immediately. The company is working with HP on a future firmware for its routers to prevent attackers from gaining control remotely and issuing commands to decrypt traffic.

Reboots, remakes, and sequels need not apply—Ars’ most anticipated games of 2016

Only original ideas allowed in this selection of upcoming titles.
Yahoo says data stolen from 1 billion accounts

by Seth Fiegerman  @sfiegerman

December 15, 2016: 4:30 AM ET
System Security: It is fun!

- System security brings together all of the fun CS topics we have learned through other courses
  - Architecture
  - Operating Systems
  - Networks
  - Compilers and Programming Languages
  - Algorithms
  - AI

- System security helps us make connections between these topics, helping us to understand them and remember them better.
What is security

Wikipedia:

Security is the degree of resistance to, or protection from, harm. It applies to any vulnerable and valuable asset, such as a person, dwelling, community, nation, or organization.
What is computer security?

- Everyone has their own definition
  - No single one is perfect

- Computer security deals with protecting data, programs, and systems against intelligent adversaries.

- Safety vs Security
  - What’s the difference between the two?
  - Do they interact?
CIA

Security is about CIA

**Confidentiality:** Keeping data and resources hidden or protected from unauthorized disclosure

**Integrity:** Data and Programs are modified in specified and authorized ways. Data integrity and origin integrity.

**Availability:** Systems and networks are available for use by legitimate users
Why is it hard?

- Security often not a primary consideration
  - Performance and usability take precedence
- Feature-rich systems may be poorly understood
- Implementations are buggy
  - Buffer overflows have been the “vulnerability of the decade” for multiple decades!
  - Cross-site scripting and other Web attacks
- Networks are more open and accessible than ever
  - Increased exposure, easier to cover tracks
- Many attacks are not even technical in nature
  - Phishing, social engineering, etc.
Why is it hard?

- It is hard to get security right because:
  - Security is hard to test for
    - Testing correctness versus security
  - It requires a deep understanding of all technologies involved in the design and implementation of a system
    - Really hard in large real systems
  - Users are typically the weakest link
  - *Asymmetry* between attack and defense
Course Focus

- Introduction to a wide range of topics in computer system and software security
  - vulnerabilities, exploit and mitigation techniques
  - malware trends and defenses against untrusted code
  - binary analysis, reverse engineering and forensics
  - software vulnerability scanning techniques and tools

- Cultivate the “security mindset”
  - Understand the modus operandi of attackers: find vulnerabilities, subvert protections, bypass defenses, . . .

- Hands-on assignments in exploit development and mitigation

- Get a taste of security research through a project (may be optional)
Ethics and Legal Considerations

- Play Fair
- Cannot teach defense without offense, but:
  - Breaking into systems is illegal!
  - Unauthorized data access is illegal!

- Computer Fraud and Abuse Act (CFAA)

- Practice on your own systems or controlled environment

- Scanning/penetration testing/etc. of third-party systems may be allowed only after getting permission by their owner
Code of Conduct

- The work that you present as your own should be your own
- Cite the resources that you used (other people’s code, documents, etc.)
- Don’t allow your code/paper summaries to be copied
- Don’t copy other people’s code or paper summaries
- Anything short of the above, will be grounds for immediate “F” grade and further disciplinary action
Some slide contents in this lecture and future ones are courtesy of Nick Nikiforakis and Michalis Polychronakis
Security

- Communication security
  - security of data channel
  - typical assumption: adversary has access to the physical link over which data is transmitted
  - cryptographic separation is necessary

- System Security
  - security at the end points
  - information cannot be encrypted, as it needs to be accessed by applications on the end system
  - logical separation is typically the basis
Security Concerns

(a) Normal flow

(b) Interruption

(c) Interception

Availability

(d) Modification

Confidentiality

Nonrepudiability

Authenticity, integrity
How to achieve security

❖ Basis is separation
  ▪ Separate adversarial entities

❖ How to separate adversaries?
  ▪ Physical separation
  ▪ Temporal separation
  ▪ Cryptographic separation
  ▪ Logical separation

❖ Security vs Functionality
  ▪ Controlled sharing
Cryptography

- Encode the data in a manner that makes it accessible only to authorized parties
  - Encryption algorithm
  - Encryption key

- Why it is not a good idea to rely on secrecy of algorithm
  - Hard to develop good encryption algorithm
  - Does not scale beyond a few users
  - Security by obscurity

- Key point: need to preserve secrecy of key
Key concepts and terminology

- Plaintext ("unencrypted")
- Ciphertext ("encrypted")
- Encryption ($E_k(X)$) Vs Decryption ($D_k(X)$)
- Key Vs Algorithm
- Cryptanalysis: Discover $k$, $X$ or both
<table>
<thead>
<tr>
<th>Type of Attack</th>
<th>Known to Cryptanalyst</th>
</tr>
</thead>
</table>
| Ciphertext only        | • Encryption algorithm  
                         • Ciphertext to be decoded                                                                            |
| Known plaintext        | • Encryption algorithm  
                         • Ciphertext to be decoded  
                         • One or more plaintext-ciphertext pairs formed with the secret key                                  |
| Chosen plaintext       | • Encryption algorithm  
                         • Ciphertext to be decoded  
                         • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key |
| Chosen ciphertext      | • Encryption algorithm  
                         • Ciphertext to be decoded  
                         • Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key |
| Chosen text            | • Encryption algorithm  
                         • Ciphertext to be decoded  
                         • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key  
                         • Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key |
Steganography

- Hiding presence of information
- Use normal-looking messages/pictures that conceal secret data
- Useful if communication is monitored for “suspicious content” by someone
- Also used for copyright protection
  - Watermark: invisible data encoded in messages that is retained in copies, and is robust in the face of typical image transformation operations
Symmetric Crypto

Secret key shared by sender and recipient

Transmitted ciphertext

Secret key shared by sender and recipient

Encryption algorithm (e.g., DES)

Decryption algorithm (reverse of encryption algorithm)

Plaintext input

Plaintext output
Model of Symmetric Crypto
Stream and Block Ciphers

- **Stream cipher**: used to encrypt digital streams of data, one bit or a byte at a time

- **Block cipher**: data is partitioned into blocks (typically 64 or 128 bits), and encryption operates on these blocks.

- **Stream ciphers can be constructed from block ciphers**
  - For this reason, crypto algorithms are developed almost exclusively for block ciphers
Structure of Symmetric Crypto

- Needs to produce a reversible mapping that maps 64-bit blocks onto other 64-bit blocks
- Good ciphers are based on Shannon’s concepts of “diffusion” and “confusion”
  - Diffusion: disperse bit-patterns within each block of data
  - Confusion: “mix-up” the order of bits within a block. In practice, use permutations specified by a key.
- In principle, good ciphers can be implemented using a table of mappings
  - Encryption key selects which mapping to use
  - Approach impractical for all except smallest block sizes
- Feistel structure: a way to build more complex ciphers from simpler ones
Symmetric Crypto Algorithms

- **DES**
  - Not considered very secure (key length of 56 bits)
- **Triple DES with two keys** (128 bits)
- **AES** (128 bits)
- **IDEA** (128 bits)
- **Blowfish** (up to 448 bits)
- **RC5** (up to 2040 bits)
- **CAST-128** (40 to 128 bits)
- **RC2** (8 to 1024 bits)
Public Key (Asymmetric) Crypto

- Uses one key for encryption and another one for decryption
  - Requires that it be computationally infeasible to compute one of the keys based on the other
- One of the two keys is private to a principal; the other key can be freely distributed to any one
  - Each principal generates his/her own pair of public/private keys, and the private key need not be revealed to any one.
- Some public key algorithms (e.g., RSA) permit both keys to be used for encryption and decryption
  - What is encrypted with one key can be decrypted with the other
Encryption in Public Key Crypto

Diagram:
- Plaintext input
- Encryption algorithm (e.g., RSA)
- Transmitted ciphertext
- Decryption algorithm (reverse of encryption algorithm)
- Plaintext output

Key elements:
- Alice's public key ring
- Bob's public key
- Bob's private key
- Joy, Mike, Ted
Authentication in Public Key Crypto

Plaintext input → Encryption algorithm (e.g., RSA) → Alice’s private key

Transmitted ciphertext

Decryption algorithm (reverse of encryption algorithm) → Plaintext output
Encryption Vs Signing

- When the encoding operation is performed using someone’s public key, the results are accessible only to that person
  - This operation can be used to ensure confidentiality of data --- hence called “encryption”
- When the encoding operation is done using someone’s private key, the results are accessible to every one.
  - But one can be sure that the message came only from the person whose public key is used for decoding --- hence called “signing”
RSA Algorithm

- Alphabet = \{0,...,n-1\}
  - in practice, \{0,...,2^k\} for \(2^k < n \leq 2^{k+1}\)

- Encryption: C = M^e \mod n

- Decryption: M = C^d \mod n
  \[= (M^e)^d \mod n\]
  \[= M^{ed} \mod n\]

- Need: \(M^{ed} = M \mod n\)

- Both sender and receiver know n.
- Sender knows e, while only the receiver knows d.
- ie, KU = (e, n), KR = \{d, n\}
RSA Algorithm Requirements

- It is possible to find $d, e, n$ s.t. $M^{ed} = M \mod n$, for all $M < n$
- It is easy to calculate $M^e$ and $C^d$
- It is infeasible to determine $d$ from $e$
### RSA Algorithm

**Key Generation**

Select \( p, q \) \( \rightarrow p \) and \( q \) both prime

Calculate \( n = p \times q \)

Calculate \( \phi(n) = (p - 1)(q - 1) \)

Select integer \( e \)

\( \gcd(\phi(n), e) = 1; \ 1 < e < \phi(n) \)

Calculate \( d \)

\( d = e^{-1} \mod \phi(n) \)

Public key

\( KU = \{e, n\} \)

Private key

\( KR = \{d, n\} \)

---

**Encryption**

Plaintext:

\( M < n \)

Ciphertext:

\( C = M^e \mod n \)

---

**Decryption**

Ciphertext:

\( C \)

Plaintext:

\( M = C^d \mod n \)
Conventional Vs Public Key Crypto

❖ Conventional crypto is fast
  ▪ Software implementations on current PCs can perform encryption at the rate of few MB/s

❖ Public key crypto is much slower
  ▪ At least 3 orders of magnitude slower

❖ Key distribution is easier with public keys
  ▪ Need to ensure authenticity of public keys
  ▪ For conventional keys, confidentiality is needed

❖ Solution
  ▪ Use conventional crypto for encrypting bulk data
  ▪ Use public key crypto to exchange keys for such encryption. • conventional keys are encrypted using public keys and sent to the recipient.
  ▪ Use certificates and certification authorities (CAs) to establish authenticity of public keys
Uses of Random Numbers

- Nonces (to protect against replay attacks)
- Session key generation
- RSA key generation

- Need cryptographically strong random number generator
  - Not enough if we had “random” numbers in a statistical sense
  - Need unpredictability
Pseudorandom number generators

- **Linear congruential method**
  - $X_{n+1} = (aX_n + x) \ mod \ m$
  - Not good for crypto applications, as it is predictable

- **Cyclic encryption**
  - $E_k(n)$, for $n = 0, 1, 2, \ldots$
  - Problem: what happens after a system restart? Does $n$ go back to zero? If so, the random number sequence becomes predictable (seen before)

- **Solutions:**
  - use something like a real-time clock plus sequence number
  - use “true random” source
Natural Random Noise

- Best source is natural randomness in real world
  - radiation counters
  - radio noise
  - Keystroke intervals
  - Network packet arrival characteristics
Miller-Rabin Test

- Pick an odd number \( n \), note \( n-1 = 2^k q \), where \( q \) is odd
- Pick a number \( 1 < a < n-1 \), compute \( a^q, a^{2q}, \ldots, a^{2^kq} \)
- If \( n \) is prime, by Fermat’s theorem
  \[
  a^{2^kq} \mod n = a^{n-1} \mod n = 1
  \]
  Hence, for some \( 0 \leq j \leq k \), \( a^{2^j q} \mod n = 1 \)
  - Case 1: \( j = 0 \): this means \( a^q \mod n = 1 \)
  - Case 2: \( j > 0 \), \( a^{2^{j-1}q} \mod n \neq 1 \), \( a^{2^j q} \mod n = 1 \)
    i.e., \( (a^{2^{j-1}q} - 1) \times (a^{2^{j-1}q} + 1) \mod n = 0 \)
    Since the first factor is nonzero, we have
    \( (a^{2^{j-1}q} + 1) \mod n = 0 \), or, \( a^{2^{j-1}q} \mod n = n-1 \)
- The algorithm tests for case 1 or case 2.
  - If the test fails, that means \( n \) is composite
  - If it succeeds, \( n \) is not guaranteed to be prime
    - but the probability of success for a nonprime is less than 0.25
- Repeat the test for \( t \) different \( a \)’s to get a prime with probability \( 1-(0.25)^t \)
Digital Signatures

- **Required properties**
  - receiver can verify who sends
  - sender can not repudiate
  - receiver can not generate

- **Conventional crypto is not very useful**
  - Sender and recipient share key, so nonrepudiability is a problem

- **Public-key signature**
  - Originator simply encrypts the message using private key
  - When the receiver gets the message decrypted using the originator’s public key, then we can be sure about who sent the message

- **Note that the encrypted message can be produced only by the originator, so all of the above properties are satisfied.**
Message Digests

- Encrypting the whole message for signature purposes is impractical (too inefficient)

- Solution
  - use one-way hash functions: compute a fixed-size (e.g., 128-bit) hash on the message
  - Encrypt the hash using private key

- One-way hash code:
  - Given P, it is easy to compute H(P)
  - Given H(P), it is impossible find P
  - No one can generate two messages that have the same message digest

- Common hash functions
  - MD5
  - SHA-1
  - RIPEMD-160
Digital Certificates

- Certificates are issued by a CA
- Every one knows the public keys of the CA
- A certificate for a principal A is simply A’s public key that is encrypted with CA’s private key
  - Only the CA could have produced such a message, so the recipient of the certificate knows that the CA vouches for A’s public key
  - If the recipient trusts CA, then the certificate provides a simple way to authenticate the public key of A.
Public-Key Certificates

- certificates allow key exchange without real-time access to public-key authority
- a certificate binds identity to public key
  - usually with other info such as period of validity, rights of use etc
- with all contents signed by a trusted Public-Key or Certificate Authority (CA)
- can be verified by anyone who knows the public-key authorities public-key
Authentication
Identity and Authentication

• Access rights granted on the basis of identity of the entity performing access (principal)
• Authentication mechanisms used to establish that a principal is who he/she claims to be
  • Alternatively, one may be interested in proving that they have certain rights

• Covers
  • User authentication
    • Main focus in the next few pages
    • Primary problem within single administrative domain where “the system” is trusted, but users are not
  • Authentication between systems
    • Primarily in the context of networked system, i.e., multiple domains with limited trust between them
Evolution of Password Schemes

• Early systems (1960-) stored plaintext passwords
  • Frustrated by hackers that were able to get to this file

• UNIX (1970s): store only hashes of passwords
  • Hash: one way function that is infeasible to revert
  • Originally used DES, subsequently shifted to MD5
    • MD5 now considered weak for this purpose, use SHA-512 or bcrypt
  • Use of salt to thwart offline dictionary attacks
    • Salt = different random value for each user, used in hashing; stored together with hashed password
Issues in Password-based Authentication

- Confidentiality of stored passwords
  - Difficult to protect stored passwords
    - Accidental disclosures (temporary copies left behind, accidental misconfiguration of file permissions)
  - Motivated attacks on a high-value target
  - Illicit copies made by system staff
  - Stealing from backup tapes

- Solution
  - Don’t store plaintext passwords
  - Original proposal: store $\text{DES}_{\text{Password}(0)}^{25}$
    - More recently, use hashes (MD5crypt, SHA-512crypt)
  - For authentication, apply same process to user-supplied password, compare with stored value (in /etc/passwd)
Categories of Attacks on Passwords

- Offline attacks: attacker has access to hashed passwords
  - Can make an unbounded number of attempts at guessing the password
    - guess, hash, compare with the hashed password
  - Brute-force attack
    - Guess password, hash, compare
  - Dictionary attack
    - Use an intelligent algorithm to enumerate passwords
    - In early days, this meant English dictionary or phone books
- Online attacks: no access to hashed passwords, so each attack attempt requires entering the password at the password dialog
  - Systems limit number of attempts, so online attacks need to succeed within a few attempts.
Password weaknesses [Morris, Thompson 79]

• In a collection of 3,289 passwords:
  • 15 were a single ASCII character
  • 72 were strings of two ASCII characters
  • 464 were strings of three ASCII characters
  • 477 were strings of four alphanumerics
  • 706 were five letters, all upper-case or all lower-case
  • 605 were six letters, all lower-case
  • 492 in various common dictionaries
• 86% of the 3,289 passwords were thus easy to crack
  • Cracked in seconds in some cases, and 100 hours in the best case --- on computers of the 70s.
Password weaknesses [www.troyhunt.com]

- Use of weak passwords is largely unchanged
- OK, there are almost no passwords of length < 4
Password weaknesses
[www.troyhunt.com]

Character type exclusivity

- Lowercase only: 50%
- Numbers only: 45%
- Uppercase only: 4%
- Other: 1%
Password weaknesses

Password reuse across Sony and Gawker

- Identical password: 67%
- Unique password: 33%
Password weaknesses
[www.troyhunt.com]
Sony passwords reused at Yahoo! Voices
Easy-to-remember passwords rely on patterns or algorithms that can be used to generate a candidate list.

- Dictionary can also be built from passwords stolen from other sites.
Password weaknesses [Gosney 12]

- Brute-force, dictionary attacks greatly speeded by GPUs

Performance of 25 AMD-Radeon GPU powered system
Password weaknesses [Gosney 12]

- Even GPUs are not too fast for some hash algorithms

Performance of 25 AMD-Radeon GPU powered system

- md5crypt
- bcrypt (05)
- sha512crypt

77 M/s

71 k/s

364 k/s
Defending against Offline attacks

• Slow down offline attacks
  • Make hash algorithm slower
  • Make attacker repeat work for every user (“salt”)
    • Each user assigned a random salt value (which is stored in the password file)
  • Original proposal: DES$^{25} \text{Password||salt}(0)$
    • Eliminates attacks that hash once, compare against passwords of all users

• Protect password file
  • /etc/passwd is world-readable, so easy to steal
  • Modern UNIX versions separate password hashes into an /etc/shadow that is readable only by root
Online attacks

• Guessing is typically unsuccessful except for the most easily guessed password
  • Delays: remove login prompt after 3 failed attempts
  • Increase delay (e.g., double) after additional failures
• Lock outs: prevent user from login after N failures
• CAPTCHAs: make user solve CAPTCHA after N failures

• Password stealing is the most viable approach for succeeding in online attacks
  • Network sniffers (solutions discussed later)
  • Phishing (fake password dialogs)
  • Keyloggers and other malware
  • Password reset
Password Theft and Trusted Path

• How to make sure that your password is not stolen when it is used
  • Key challenge today due to spyware, spoofing, phishing, etc.
• Trusted path: a secure way for a user to communicate with the subsystem performing user authentication
  • Ctrl-Alt-Del on Windows
    • Provided that the OS is not infected ...  
      • And the BIOS is not infected ...  
      • And the hardware is not malicious ...
Phishing and Trusted Path

• Phishing attacks typically involve tricking a user into revealing their passwords
  • Attacker sets up a web site that looks like attack target, e.g., a bank web site
  • Attacker steals the password when the user tries to log into the fake web site
Phishing Defenses

• Two-stage login with personalized prompts
  • Security skins, site-keys (personalized images)
    • Requires user vigilance
      ▪ Phisher may say “system failure, so we can’t retrieve your image at this time”
      ▪ Small “key space” for possible images

• Security questions
  • pain to use
  • small key space
  • answers easily guessed, especially by family/friends
Phishing Defenses

• SSL provides strong defense (completes trusted path)
  • people lulled into accepting self-signed certificates
  • But today’s browsers provide stronger warning (or silently suppress) sites that change a CA-provided certificate into a self-signed one
• social engineering (“our SSL servers are down today”)
• DNS redirects!
• Compromise of Certification Authorities
  • Once thought unlikely, but is increasingly being used against high-value targets
Summary of Password weaknesses

- **Offline**
  - Brute-force and dictionary attacks greatly speeded up by GPUs
  - Dictionary attacks speed up the search, especially if they are based on passwords revealed in data breaches

- **Online and offline:**
  - Use of weak passwords
  - Keyloggers (and formerly, network sniffers)
  - Social engineering (phishing)
  - Password reset mechanisms
More password problems

- Easy-to-remember passwords may be easy to guess
  - Dictionary attacks
- Password management
  - Dealing with multiple passwords
  - Writing passwords down (should I?)
  - Password selection rules
  - Password expiry rules
Password weaknesses: Non-solutions

• CAPTCHAs to defeat guessing attacks
  • Increasingly, becoming too hard for humans!
• Security questions
  • Often, answers are available on social media
• Password rules
  • A nightmare for users
  • Questionable increase in password strength
    • Users often add easily guessed prefix or suffix to a simple password, e.g., “0-” or “#1”
• Alternative password schemes
  • Face or picture recognition
Improving basic password schemes

• Using master password
  • Generate random passwords, encrypt them using master password, store them
• One-time (single-use) passwords (OTP)
• Biometrics (?)
• Visual passwords (??)
• Two-factor authentication: Require two forms of authentication
  • Password + small device or smartcard
  • Password + biometrics
  • Password + OTP sent by email or text
  • Relies on authentication needed to access email/text
Using Master Passwords

• A master password is used to encrypt all other passwords
  • Focus on creating/remembering one strong password
    • low tech approach: all other passwords written down in a file that is manually encrypted with the master password
    • more usable approaches rely on “password managers”
  • built into common applications
    • ssh
    • Browsers
Password managers on browsers

• Benefits
  • Allows strong passwords unique to each website
    • Generate a random password for each site
  • Reduces theft due to practices such as writing them down
  • Computers are not easily phished
    • Avoids password being revealed to sites that
      ▪ look similar
      ▪ have URLs that are misspelled or have typos
      ▪ use http instead of https
  • Immune to keyloggers and malware snooping on cut/paste buffers
    • But key loggers can capture your master password

• Drawbacks
  • Bad idea on shared devices
  • False sense of security if master password can be stolen
Authentication across the network

• Trust client to authenticate (avoid network transmission of password)
  • Host-based authentication
    • Used in NFS, also rsh/rlogin/rexec with hosts.equiv
    • Not a great option today, as users often have admin privileges on client machines

• Server-side authentication of plaintext passwords
  • Don’t trust client computer; server performs this task
  • Used by rsh/rlogin/rexec, telnet, ftp, etc.
  • Bad option unless you trust all clients on the network
    • Otherwise, easy password compromise by network sniffers
Authentication across the network

• Trust client to encrypt user-supplied password
  • The encryption part is performed by the client, while the checking part is done by the server
  • Only encrypted password transmitted over network
    • But it is as good as unencrypted password!
      ▪ A rogue client can sniff and reuse this encrypted password to log into the server, without ever needing to decrypt it

• Solutions against such replay attacks
  • One-time passwords (theft no longer a problem!)
  • Challenge-response protocols (esp. using public keys)
One-time passwords

• Start with a password P to generate a sequence of one-time passwords $O_1...O_N$
  • Requirements: $O_k$ should not provide any info about $O_{k+1}, O_{k+2},...,O_N$
• Solution: $O_k = H^{N-k}(P)$, where H is a secure one-way hash function

• Protocol:
  • System $\rightarrow$ User: i
  • User $\rightarrow$ System: $H^{N-i}(P)$
  • Even if user doesn't respond, use i+1 as next challenge

• Note: system need not store P, just the previous OTP
  • check that $H$(current OTP) = prev OTP
Challenge-response protocols

• SSH
  • Password based authentication
    • S → C: KU
  • C → S: E_{KU}(K_{SES} = \text{random}()), E_{K_{SES}}(password)
  • All subsequent communication encrypted using K_{SES}
  • Problems: integrity of KU not assured. SSH asks user to confirm the key the first time a server is accessed, and saves the key for use in future accesses to same server
  • Public key based authentication
    • C → S: KU
    • S → C: Verify presence in \sim\text{user}/.ssh/authorized_keys, send challenge = E_{KU}(random)
    • C → S: decrypt and send the result
Challenge-response protocols

- Web sites use password authentication over https
  - $S \rightarrow C$: Public key certificate $E_{KR_{CA}}(KU_S)$
  - $C \rightarrow S$: $E_{KU_S}(K_{SES} = \text{random})$
- All subsequent communication encrypted using $K_{SES}$
- Similar to SSH password authentication
- Protocols such as telnet can be made secure by simply carrying their traffic over https
- Challenges
  - Certificates cost $\$, so there were self-signed certs
    - Users got used to certificate violations, ignored warnings
    - Recently, certificates are available for free, so this problem is gradually disappearing
  - Recent browsers make it difficult to ignore warnings
    - Some violations silently disallowed, e.g., changes to certificates of certain servers
Two-factor authentication: SecureID

- A hand-held device sold by RSA
  - Widely deployed in enterprises
  - Well-publicized hack on this system in early 2011 led to attacks on high-profile businesses
- Uses a device-specific secret to generate authentication token every minute or so
  - E.g., AES_{K_s}(Time)
  - Tamper-resistant device, so one cannot steal $K_s$
  - Server must know device-specific secret
- Combined with a PIN or password
Summary of User Authentication Approaches

- **Something you know**
  - A secret key (password)
  - Issues: difficulty of guessing, ease of remembering

- **Something you have**
  - key, magnetic card, RFID chip, smart card, cell phone, ...
  - Issue: possibility of losing
  - Combine with a secret to minimize damage due to loss

- **Something you are**
  - Fingerprint, photo, voice, handwriting, ...
  - Issues: accuracy of recognition, possibility of stealing
  - Works best in a supervised setting
Biometrics

• Authenticate by recognizing some aspect of human physiology, anatomy, skill or trait
  • Physiological (fingerprint, iris, retina, face, hand geometry, DNA)
  • Behavioral (keystroke, voice/speech, ...)

• Benefits:
  • convenience
  • protection against poor choice of passwords
  • more difficult to steal, particularly in controlled (supervised) setting

• Drawbacks
  • Need for special equipment
  • Not 100% reliable (false positives and negatives)
  • User acceptance
Biometrics: Terminology, Issues

- False match or acceptance rate (FMR/FAR)
  - “fraud rate”
- False non-match/rejection rate (FNMR/FRR)
  - “insult rate”
- trade-off between the two: equal error rate
- verification (pair-wise comparison) Vs identification (one-to-many comparison)
  - even very small error rates get magnified for the latter, and hence become unacceptable.

Issues

- User acceptance
- Privacy and discrimination
- Can’t be canceled/changed if stolen
- Danger of physical harm to owner
Handwritten signatures

• Routinely used in transactions and contracts for centuries
• Recognition may be manual, machine-assisted or completely mechanical
• Different approaches may be warranted based on application
  • legal Vs check-out counter Vs check-clearing for small checks
• Signature tablets
  • record signature dynamics as well as the resulting image
Fingerprints

- most commonly used biometric

Issues:
- even low error rates can compound when doing a one-to-many match
- manipulation: lift prints artificially and deposit where there are needed.
- ++ mature
- ++ as always, deterrent effect can be higher than actual effect
Iris recognition

- **Benefits**
  - unique for each person
  - does not wear out or is exposed to external environment
  - easy to make out from a picture.
  - many times the number of degrees of freedom as fingerprint
  - minimally influenced by genetics
  - stable through lifetime

- **Gabor filters** -- a signal processing technique to transform an image of the iris into a 256-byte code. Two codes computed from same iris will match in 90% of the bits
  - Compare with fingerprints, where detection, classification and orientation of minutiae is hard.

- **Can achieve very high accuracy in controlled settings, but real-world performance not as good**

- **Other issues:**
  - Requires camera-to-eye distance of approx. 2ft or less (intrusive)
  - Can potentially be copied
Voice Recognition

• text-dependent recognition (challenge-response)
• noise can be a problem (may need microphone held close to mouth)
• one-to-many comparisons are not very accurate
• affected by stress, cold, alcohol or other drugs, ...
Other

- Keystroke dynamics
- Hand geometry
- Hand-drawn pictures
- Retina
- DNA
Problems with Biometrics

- age of reference data (e.g., fingerprint)
- age of data (when was that fingerprint left? yesterday when the bank robbery took place, or last week when there was a legitimate visit to the bank?)
- recordings
- collusions (voluntarily provide bad writing samples or photos)
- birthday problem
- combining biometrics does not necessarily help: it may reduce false accepts, but at the cost of increased false rejects (or vice-versa)
- may not work for all users ("goats")
- objections based on social and religious concerns
Visual Passwords

• Leverage highly evolved visual perception
  • Pictures seem so much easier to remember than the details in an arbitrary text password

• Several schemes
  • Passpoints: select points on an image
  • Select images from an array
    • Passfaces: leverage human capacity to recall faces
    • Random art
    • Concrete nouns
Issues with Graphical Passwords

- Many of the basic attack techniques continue to work
  - Dictionary attacks, guessing, social engineering, ...
- Shoulder-surfing
- Entropy
  - User studies have revealed that users tend to favor some images over others, e.g., pretty faces of people from one’s own race
- Memorability has not been conclusively demonstrated
Password weaknesses: Solutions

• Password managers, master passwords
  • Often thwarted by lawyers and administrators
• Public keys, e.g., SSH or PGP
• Two-factor authentication
  • Tokens, cards, biometrics, ...
• One-time passwords or PINs
  • Especially useful if a channel trusted by both sender and receiver is always available, e.g., SMS
Summary of User Authentication

• Purpose: bind physical-world entities with cyber-world entities
• Means: Present “credentials”
  • Secret
    • passwords
  • Possession
    • Key-card
    • Biometrics
• Attacks: theft, guessing attacks,…
• Defenses
  • Multi-factor authentication
  • Password managers
  • One-time passwords
Security Policies and Enforcement Mechanisms
Terminology and concepts

- Principals, Subjects, Objects
- Principle of least privilege
  - Throughout execution, each subject should be given the minimal access necessary to accomplish its task
  - Needs mechanisms for rights amplification and attenuation
- Reference monitors
  - Abstract machine that mediates all access
- Security kernel
  - Hardware, firmware and software elements that implement the reference monitor
- Trusted Computing Base
  - Totality of protection mechanisms in the system
  - Smaller TCB => Greater assurance that the system is secure
Overview

- **Access control**
  - Mandatory Vs Discretionary policies

- **Capabilities**

- **Information flow**

- **Least privilege principle**
  - Domain and type enforcement (DTE)
  - POSIX Capabilities

- **Other policies**
  - Chinese wall
  - Clark-Wilson

- **Policies for containing untrusted code**

- **Manageability**
  - Role-based access control (RBAC)
  - Deletation and trust management
Access control

- Typically, three kinds of entities
  - User (principal)
  - Subject: typically, a process acting on behalf of user
  - Object: files, network sockets, devices, …

- Goal: Control access to *operations* performed by *subjects* on *objects*
  - Examples of operations
    - Read
    - Write
    - Append
    - Execute
    - Delete
    - Change permission
    - Change ownership
Discretionary Access Control

- Permissions specified by users
  - permission on an object is set by its owner
  - typical on most OSes (UNIX, Windows, ...)

- Represented using a matrix
  - Indexes by subject and object
  - Each element specifies the rights available to subject on that object (read, write, etc.)
  - Implementations
    - ACL (associated with an object, represents a column)
    - Capabilities (associated with subject, represents a row)

- Improve manageability using indirection
  - Groups
  - Roles (RBAC)
  - Inheritance
  - Negative permissions
Implementation of DAC on UNIX

- All resources are “files”
- Each file has a owner and group owner
- Permissions divided into 3 parts
  - For owner, group, and everybody else
  - 3 bits per part: read/write/execute
- Subjects inherit the userid of parent
  - Programs that perform user authentication need to set this info
  - Exception: setuid programs (privilege delegation/amplification mechanism)
    - Suid and sgid bits
- No permission checks on superuser (userid 0)
  - Permission checks based on userid --- usernames used only for login
- Defaults (umask)
- Changing permission
- Changing ownership
- Recent additions
  - Access control lists
  - Sticky bit
Effective, Real and Saved UID/GID

- **Effective**: the uid used for determining access privileges
- **Real**: the “real” user that is logged on, and on whose behalf a process is running
- **Saved**: allows processes to temporarily relinquish privileges but then restore original privileges
  - When executing a setuid executable, original euid is saved (or it could be explicitly saved)
  - Setting userid to saved userid is permitted
DAC on Windows Vs UNIX

- **OO-design: permissions can differ, depending on type of object**
  - NTFS files offer additional rights: delete, modify ACL, take ownership
    - Files inherit permission from directory
  - Use of Registry for configuration data
    - Richer set of access permissions for registry entries (e.g., enumerate, create subkey, notify, …)

- **Mandatory file system locks**

- **No setuid mechanism**
Capabilities

- “Tickets” to gain access to a resource
  - Combine objects and access rights into one package
  - Must be unforgeable
  - Transferable

- Examples
  - Passwords and cryptographic keys
  - Certificates
    - Anything cryptographically signed can be thought of as a capability
  - File descriptors
    - Handles to information maintained within OS kernel
  - Some cookies (e.g., session cookie) in web applications
Capabilities

- **Capabilities in their purest form not widely used in OSes**
  - More difficult to implement than ACLs
    - Need forever unique object ids that don’t change
    - Need to use crypto or rely on OS primitives that may be hard to realize
  - Difficult to manage
    - How do we determine the permissions held by a user?
    - *Do we want to allow them to pass around their capability? What about theft?*
    - How long do we store them?
    - How can we revoke permissions?

- **Provide a better framework than ACLs when policy enforcement is NOT centralized**
  - Kerberos uses capabilities for access across hosts
    - Uses capabilities with different time scales
    - Accesses within a host typically based on ACL mechanism of host OS
  - Web applications use cookies containing session ids to indicate when a user has successfully authenticated
## Access Control Matrix

<table>
<thead>
<tr>
<th></th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Alice</strong></td>
<td>r,w</td>
<td>w</td>
<td>r</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Bob</strong></td>
<td>r,w,x</td>
<td>r,w</td>
<td>-</td>
<td>r,w</td>
<td></td>
</tr>
<tr>
<td><strong>...</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Indexes by subject and object**

- Each element specifies the rights available to subject on that object (read, write, etc.)

**Implementations**

- **ACL** (associated with an object, represents a column)
  - **O1**: Alice:rw, Bob:rwx, ...
  - **O4**: Alice:-, Bob:rw, ...

- **Capabilities** (associated with subject, represents a row)
  - **Alice**: O1:rw, O2:w, O3:r, O4:-, ...
  - **Bob**: O1:rwx, O2:r, O3:-, O4:rw, ...
Mandatory Access Control (MAC)

DAC Limitations

- “Trojan Horse” problem: assumes that users are in full control of the programs they execute
  - What if a program changes permissions without user’s knowledge?
- Provides no protection if a resource owner did not bother to set the ACL properly

To overcome these problems, MAC moves the responsibility to a central point, typically the system administrator

- Organizations want to control access to their resources
- Don’t want to rely on individual employees to ensure that organizational policies are enforced
Motivation for MLS

Access control policies do not provide any way to control the manner in which information is used:

- once an entity is given access to some information, it can use this information in any way
  - Can share it with any one

MLS policies control information flow, and hence control how information is used

Developed originally in the context of protecting secrets in the military
MLS: Confidentiality Policies

- An object is labeled with a level L
  - Labels correspond to points in a lattice
  - Typical levels used in military include:
    - unclassified, classified, secret, top secret

- A subject is associated with a clearance level C
  - A subject can access an object if his clearance level is equal to or above the object’s level

- Information is also compartmentalized
  - “Need-to-know” principle is used to decide who gets to access what information
    - e.g., top-secret information regarding nuclear fuel processing is made available to those working on nuclear-related projects
MLS: Bell-LaPadula Model [1973]

To ensure that sensitive information does not leak, we need to ensure:

- No “read-up:”
  - A subject S can read object O only if C[S] >= L[O]

- No “write-down:”
  - A subject can write an object O only if C[S] <= L[O]
  - Prevents indirect flows where a top-secret-clearance subject reads a top-secret file and writes to a secret file, which may then be read by someone with a lower (i.e., secret) clearance

- Based on the idea that any subject that reads information at a certain level has the potential to leak information at that level whenever it outputs anything.
MLS: Biba Model (Integrity)

- Designed to ensure integrity rather than confidentiality
  - In non-military settings, integrity is more important

- Conditions
  - No “read-down:”
    - A subject S can read object O only if C[S] <= L[O]
    - A subject’s integrity can be compromised by reading lower integrity data, so this is disallowed
  - No “write-up:”
    - A subject can write an object O only if C[S] >= L[O]
    - The integrity of a subject’s output can’t be greater than that of the subject itself.

- Variation: Low Water-Mark Policy (LOMAC)
  - Allow read-downs, but downgrade subject to the level of object

- Both policies ensure system integrity
Problems with Information Flow

- **In a nutshell: difficult to use**
  - “Label creep:” More and more objects become sensitive, making it difficult for the system to be used by lower-clearance subjects
  - Exceptions need to be made, e.g., an encryption program
    - “Trusted” programs are allowed to be exempted from “*”-property
    - But exceptions are misused widely, since it is hard to configure whole systems carefully so that “*”-property can be enforced without breaking functionality

- **Motivate alternate approaches, or hybrid approaches**
Alternative Approaches

✦ Key goal: Mitigate damage that may result from all-powerful root privileges
  ▪ Break down root privilege into a number of sub-privileges
  ▪ Decouple user privileges from program privileges

✦ Examples
  ▪ Domain and type enforcement
    ▼ SELinux
  ▪ “Linux capabilities”
    ▼ Different from classical notion of capabilities described earlier under DAC
Domain and Type Enforcement

- **Subjects belong to domains**
  - Users have default domains, but not all their processes belong to the same domain
    - Some processes transition to another domain, typically when executing another program

- **Objects belong to types**

- **Policies specify which domains have what access rights on which types**
  - Enable application of least-privilege principle
  - Example: a media player may need to write its configuration or data files, but not libraries or config files of other applications

- **Domain transitions are an important feature**
  - Can occur on exec, as specified by policy
DTE and SELinux

- Security-enhanced Linux combines standard UNIX DAC with DTE
  - Note: SELinux also supports other MAC mechanisms (e.g., MLS) but these are typically not enabled/configured
- Intuitively, the idea is to make access rights a function of (user, program, object)
- Roughly speaking, MLS requires us to trust a program (and not enforce “*”-property), or fully trust it (ie it may do whatever it wants with information that it read)
  - In contrast, DTE allows us to express limited trust, i.e., grant a program only those rights that it needs to carry out its function
DTE/SELinux Vs Information Flow

- In practice DTE has turned out to be “one policy per application”
  - Scalability is clearly an issue
  - In addition, SELinux policies are quite complex
  - While DTE is able to gain additional power because it captures the fact that trust is not transitive, this very feature makes DTE policies difficult to manage
    - What overall system-wide assurances can be obtained, given a set of DTE policies developed independent of each other

- Information flow policies are simpler, and closely relate to high level objectives
  - Confidentiality or Integrity
  - But neither approach is easy enough for widespread use
Linux (POSIX) Capabilities

- **Goal:** Decompose root privilege into a number of “capabilities”
  - `CAP_CHOWN`
  - `CAP_DAC_OVERRIDE`
  - `CAP_NET_BIND_SERVICE`
  - `CAP_SETUID`
  - `CAP_SYS_MODULE`
  - `CAP_SYS_PTRACE`
  - ...

- **Differs from classical capabilities**
  - Captures access rights, but not associated with any object
  - Unforgeable only because the capabilities are never present in the subject
    - They are maintained by the OS kernel for every process, similar to how subject ownership is maintained in the kernel
Linux (POSIX) Capabilities

- **Effective, Permitted and Inheritable capabilities**
  - Somewhat related to (and guided by) effective, real and saved userids
  - Effective: accesses will be checked against this set
  - Permitted: superset of effective, cannot be increased
    - Effective set can be set to include any subset of permitted
  - Inheritable: capabilities retained after execve
    - at execve, permitted and effective sets are masked with inheritable

- **Attaching capabilities to executables**
  - Allowed: capabilities not in this set are taken away on execve
  - Forced: “setuid” like feature --- given to executable even if parent does not have the capability
  - Effective: Indicates which of the permitted bits are to be transferred to effective
Policies and Mechanisms for Untrusted Code

- **Isolation**
  - Two-way isolation
    - Chroot jails
    - Userid-based isolation
    - Virtual machines
  - One-way isolation
    - Read access permitted, but write access denied

- **System-call sandboxing**
  - Linux seccomp, seccomp-bpf and eBPF
  - Delegation

- **Information flow**
chroot jails

- **Makes the specified directory to be the root**
  - Process (and its children) can no longer access files outside this directory

- **Requires root privilege to chroot**
  - For security, relinquish root privilege after chroot
  - All programs, libraries, configuration and data files used by this process should be within this chroot’ed dir

- **Isolation limited to file system**
  - e.g., it does not block interprocess interactions
  - For this reason, chroot jail is useful mainly to limit privilege escalation; but the mechanisms is insecure against malicious code.
Userid based isolation

- **Create a new userid for running untrusted code**
  - Real user’s userid is not used, so the “Trojan horse” problem of altering permissions on user’s files is avoided

- **Android uses one userid for each app**
  - Default permissions are set so that each app can read and write only the files it owns (except a few system directories)

- **Protects against malicious interprocess interactions**
  - kill, ptrace, …

- **Better than chroot, but still insufficient against malicious code**
  - Can subvert benign processes by creating malicious files that may be accidentally consumed by them
    - Many sandbox escape techniques work this way
  - Too much information available via /proc, as well as system directories that are public: Can use this info to exploit benign processes via IPC
One-way isolation

- **Full isolation impacts usability**
  - untrusted applications are unable to access user’s files
  - makes it difficult to use nonmalicious untrusted applications

- **One-way isolation**
  - Untrusted application can read any data, but writes are limited
    - cannot overwrite user files
    - More importantly, benign applications don’t ever see untrusted files
      - Eliminates the possibility of accidental compromise

- **Key issues:**
  - Ensuring consistent view
    - Application creates a file and then reads it, or lists the directory
    - Inconsistencies typically lead to application failures
  - Failures due to denied write permission
    - Can overcome by creating a private copy of the file

- **Both issues overcome using copy-on-write file system**

- **Note**
  - does not protect against loss of confidential data (without additional policies)
  - securing user interactions is still a challenge
System-call sandboxing: seccomp

- Seccomp is a Linux mechanism for limiting system calls that can be made by a process
  - Processes in the seccomp sandbox can be make very few system calls (exit, sigreturn, read, write).
- More secure than previous mechanisms, but greatly limits actions that can be performed by a sandboxed process
  - Useful if setup properly, e.g., in Chrome, Docker, NativeClient
- Seccomp-bpf is a more recent version that permits configurable policies
  - Allowable syscalls specified in the Berkeley packet filter language
  - Policies can reference syscall name and arguments in registers
- Unfortunately, most interesting policies are out-of-scope, as they reference data in process memory, e.g., file names
  - For this reason, seccomp-bpf is not much more useful than seccomp
System-call delegation

- Used in conjunction with strict syscall sandboxing
  - Key idea: Delegate dangerous system calls to a helper process
  - Helper process is trusted
    - It cannot be manipulated by untrusted process
    - Can implement arbitrary, application-specific access control logic
    - Avoids race conditions

- Works only if
  - System call semantics permits delegation
    - E.g., not applicable for fork or execve
  - Results can be transferred back transparently to untrusted process
    - E.g., file descriptors can be sent over UNIX domain sockets using sendmsg
Securing untrusted code using information flow

- Untrusted code = low integrity, benign code = high integrity

- Enforce the usual information flow policy that
  - Deny low integrity subject’s writes to high integrity objects
    - Prevents “active subversion”
  - Deny high integrity subject’s read of low integrity objects
    - Prevents “passive subversion”
      - fooling a user (or a benign application) to perform an action, e.g., click an icon on desktop
      - exploit a benign process, e.g, benign image viewer compromised by reading a malicious image file

- Can provide strong guarantee of integrity
  - Not subject to “sandbox escapes”

- Usability issues still need to be addressed
Commercial Policies

- High-level policies in commercial environments are somewhat different from those suitable for military environments

- Examples
  - Chinese Wall (conflict of interest)
  - Clark-Wilson

- Common principles
  - Separation of duty: critical functions need to be performed by multiple users
  - Auditing: ensure actions can be traced and attributed, and if necessary, reverted (recoverability)
Clark-Wilson Policy

- **Focuses on data integrity rather than confidentiality**
  - Based on the observation that in the “real-world,” errors and fraud are associated with loss of data integrity

- **Based on the concept of well-formed transactions**
  - Data is processed by a series of WFTs
  - Each WFT takes the system from one consistent state to another
    - Operations within a WFT may temporarily make system state inconsistent
  - While the use of WFTs guarantee consistency of system state, we need other mechanisms to ensure integrity of WFTs themselves
    - Was that a fraudulent money transfer? Was that travel voucher properly inspected?
      - Relies primarily on separation of duty
  - Auditing to verify integrity of transactions
  - Maintain adequate logs so that WFTs in error can be undone
Chinese Wall Policy

Addresses “conflict of interest”
- Common in the context of financial industry
- Regulatory compliance, auditing, advising, consulting,..

Defined in terms of
- CD: objects related to a single company
- COI classes: sets of companies that are competitors
- Policy: no person can have access to two CDs in the same COI class
  - Implies past, present or future access
Policy Management

- **Goal:** simplify the set up and administration of security policies

- **Topics**
  - Role-based access control (RBAC)
  - Administrative policies
    - Who can change what policies
  - Delegation and trust management
Roles vs groups: Essentially the *same mechanism* but *different interpretations*

- Role: a set of permissions
- Group: a set of users

Roles and groups provide a level of indirection that simplifies policy management

- Based on the functions performed by a user, he/she is given one or more roles
  - When the user’s responsibilities change, the user’s roles are updated
  - When the permissions needed to perform a function are changed, the corresponding role’s permissions are updated
  - Does not require any updating of user information
Delegation

- Ability to transfer certain rights to another entity so that it may act on behalf of the first entity
- Delegation is necessary for managing authorizations in a distributed system
  - Decentralized/distributed control

- How to implement delegation
  - The issue is one of trust and granularity
  - Multiple levels of delegation rely on a chain of trust
    - Can be implemented using certificates

- Trust management
  - Systems designed to manage delegation, and enforce security policies in the presence of delegation rules and certificates
Virtual Machines
Concepts

• Virtualization:
  – Creation of flexible substitutes for actual resources.
    • The substitutes and their actual counterparts:
      – have same functions and external interfaces
      – differ in size, performance, cost etc.

• Resources to virtualize
  – CPU
  – Memory
  – I/O
Concepts

• System Virtualization
  – System virtualization creates several virtual systems within a single physical one.

• VMM (or hypervisor)
  – Virtual machine monitor is the software layer providing the virtualization.

• VM
  – Virtual machine is the virtual systems running on top of VMM
Brief History

• 1960s, first introduced, for main frames
  – Motivation: hardware cost etc.
• 1970s, an active research area
• 1980s, underestimated
  – Multitask modern operating systems took its place
  – Decreasing in hardware cost
• late 1990s, resurgence: software techniques for x86 virtualization
  – Many applications: mixed-OS develop environment, security, fault tolerance etc.
• mid 2000s, hardware support from both Intel and AMD
Types of Virtualization

• Process virtualization (virtualize one process)
  – The VM supports an ABI: user instructions plus system calls
  – Dynamic translators, JVM, ...

• OS or Namespace virtualization (multiple logical VMs that share
  share the same OS kernel)
  – Isolates VMs by partitioning all objects (not just files) into
    namespaces
  – Linux containers and vServer, Solaris zones, FreeBSD jails, Docker

• System (or full) virtualization (whole system: OS+apps)
  – The VM supports a complete ISA: user+system instructions
  – Classic VMs, whole system emulators (and many others we discuss in
    next slides)
Architectures

- **Type I**: The VMM runs on bare hardware ("bare-metal hypervisor")

<table>
<thead>
<tr>
<th>guest application</th>
<th>guest application</th>
<th>guest application</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual-machine monitor (VMM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Architectures

- **Type II**: The VMM runs as an ordinary application inside host OS (hosted hypervisor)

<table>
<thead>
<tr>
<th>guest application</th>
<th>guest application</th>
<th>guest application</th>
</tr>
</thead>
<tbody>
<tr>
<td>guest operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual-machine monitor (VMM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host operating system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>host hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Issues in CPU Virtualization

• Protection levels
  – Ring 0 (most privileged)
  – Ring 3 (user mode)

• Requirement for efficient/effective virtualization
  – Privileged instructions
    • Trap if executed in user mode
  – Sensitive instructions
    • affect important “system state”
  – If privileged==sensitive, can support efficient “trap and emulate” approach
    • Virtualized execution = native execution+exception handling code that emulates privileged instructions

• For x86, not all sensitive instructions are privileged
  – Some instructions simply exhibit different behaviors in user and privileged mode
Virtualization Approaches

- Full virtualization using binary translation
  - Problem instructions translated into a sequence of instructions that achieve the intended function
  - Example: VMware, QEMU
Virtualization Approaches

• Paravirtualization: OS modified to run on VMM
  – Example: Xen
Paravirtualization

• No longer 100% interface compatible, but better performance
  – Guest OSes must be modified to use VMM’s interface
  – Note that ABI is unchanged
    • Applications need not to be modified

• Guest OSes are aware of virtualization
  – privileged instructions are replaced by hypervisor calls
  – therefore, no need for binary translation
Xen and the Art of Virtualization
Virtualization Approaches

- Hardware-assisted virtualization
Hardware-assisted Virtualization

• Processor
  – AMD virtualization (AMD-V)
  – Intel virtualization (VT-x)
AMD-V: CPU virtualization

• Separates CPU execution into two modes
  – hypervisor executes in host mode
  – all VMs execute in guest mode

• Both hypervisor and VMs can execute in any of the four rings

• Hypervisor can
  – explicitly switch from host mode to guest mode
  – specify which events (e.g. interrupts) cause exist from guest mode
Memory Virtualization

• Access to MMU needs to be virtualized
  – Otherwise guest OS may directly access physical memory and/or otherwise subvert VMM
• Physical Memory is divided among multiple VMs
  – Two levels of translation
    • Guest OS: guest virtual addr → guest physical addr
    • VMM: guest physical addr → machine addr
Memory Virtualization

- Shadow page table needed to avoid 2-step translation
  - When guest attempts to update, VMM intercepts and emulate the effects on the corresponding shadow page table
AMD-V: Memory Virtualization

- CPU is aware of
  - the existence of VM
  - two-level address translation

- AMD’s nested page table
  - (Intel VT-x has a similar scheme called Extended Page Table)
  - managed by VMM
  - guest physical addr -> machine addr
  - guest OS directly updates its guest page table
  - therefore, no need for a shadow page table
I/O Virtualization

• The VMM
  – intercepts a guest’s I/O action
  – converts it from a virtual device action to a real device action
Security Applications

• Honeypot systems and Malware analysis
  – VM technology provides strong isolation that is necessary to run malware without undue risks
    • Strong resource isolation: CPU, memory, storage
    • Snapshot/restore features to speed up testing and recovery

• High-assurance VMs
  – On a single workstation, can run high assurance VMs that support some security functions, but may not provide general-purpose functions
    • single-purpose VM scheme facilitates stricter security policies
    • In contrast, security policies that are compatible with the range of desktop applications being used today will likely be too permissive.
Security Applications

• Protection from compromised OSes
  – Modern OSes are too complex to secure
  – Malware-infested OS may subvert security software (virus and malware scanners)
  – Instead, rely on VMM
    • run malware and rootkit detection techniques in VMM
    • enforce security properties from within the VMM
Security Challenges

- Virtualization leads to co-tenancy
  - VMs belonging to distinct principals use the same hardware
    - Strong isolation is necessary or else attacks become too easy
      - Containers don’t offer enough security if some principals can be downright malicious
    - Even with strong isolation, provide increased opportunities for side-channel attacks
  - Denial of service is difficult to prevent
    - But often, it is not a problem in practice as bad behavior is expensive, and/or is detected and the culprit punished
Docker Security

• Isolation of containers
  – namespaces: each container cannot see entities (files, processes, pids, network interfaces, ...) in other containers
  – cgroup: enables resource accounting and limiting --- including CPU, memory, disk I/O, etc.
    • one bad container cannot use up all resources

• Container infrastructure and services (docker daemon)
  – containers can share files/directories with the host OS, but this can be very dangerous, e.g., allow root user in a container to change critical host OS files
  – administrative services (e.g., creation of containers) can be abused, so interface to docker daemon should be restricted

• Limit further using Linux capabilities
  – programs running with containers typically don’t need root privilege
  – we can use Linux capabilities to take away almost all of the power of the root
Malware
Current Threats (Fourth generation)

• Steal confidential information
  • Credit-card/bank account #s, passwords, …
  • Trade secrets and other proprietary information
  • Security-sensitive information
    • Useful for breaching physical world security

• Establish base for future operations
  • Conduit for future attacks

• Surveillance
  • Capture keystrokes, microphone or camera input
  • Reveal information about software installed
  • Snoop on web sites visited
Current Threats (Continued)

- Driven by commercialization of Malware
  - Thriving black-market for exploits
    - Zero-day exploits have arrived
  - “Bot”-centric model for cyber crime
    - Relay spam (e-mail scam, phishing)
    - Extortion (using DDoS or targeted attacks)
    - Focus on desktop (rather than server) vulnerabilities
  - Profit-driven adware and spyware
    - Customer-profiling, niche-marketing
    - IP protection (digital rights management)
    - aggressive installation, stealth (rootkits, spyware)
  - Targeted attacks on high-value targets
    - Political activists
    - International adversaries
    - People with access to valuable information
      - CEO/CFO with access to financial information on publicly traded companies
      - Researchers with access to proprietary formulas or other valuable IP
Modern Threats: A Glance

- Software
  - Viruses
  - Worms
  - DDoS and Botnet
  - Rootkits
  - Spyware

- Goal of software
  - Spam
  - Phishing
  - Online extortion
  - ...

Computer Virus

- **Properties**
  - Replicates itself
  - Attaches to other non-malicious code

- **Early versions spread via floppy disks, while recent viruses spread through the internet.**

- **Examples**
  - Boot sector virus (difficult on OS with memory protection)
  - Other OS level virus
  - Virus that attaches to programs, scripts, libraries
  - Macro virus
  - Mail attachments
  - …
Disk-based Computer Viruses

- **1982, Elk Cloner**
  - First virus in the wild
  - Targeting Apple II

- **1986, (c)Brain**
  - First virus for IBM PC
  - A boot sector virus

- **1995, Concept virus**
  - First Macro virus

- **1998, CIH**
  - One of the most harmful widely circulated viruses
  - Overwrites both hard disks (data loss) and Flash BIOS (hardware damage)
Macro Virus

• Written in a macro language.
• Macros can perform operations that the software can do.
• Often, a simple solution: turning off the macro feature
CIH Virus

- Spreads via Portable Executable files under Windows 95/98/Me.

- Damages:
  - Overwriting the first 1024KB of the hard drive with zeroes ➔
    Loss of data on the entire hard drive
  - Overwriting the Flash BIOS with junk code ➔
    Computers cannot boot any more

- Activated in the public eye on April 26, 1999
- An untold number of computers worldwide were affected, much in Asia
Melissa

- Found on March 26, 1999
- Targetting Microsoft Word and Outlook-based systems, and creating considerable network traffic
- Shut down many Internal mail systems
  - That got clogged with infected e-mails propagating from the worm
- Inside a file called “List.DOC”
- Can mass-mail itself from email client Microsoft Outlook 97 or Outlook 98.
- Attempts to mass mail itself once an infected Word document is opened.
ILOVEYOU

• First appeared on May 3, 2000
• Caused widespread e-mail outages, an estimated $10 billion in economic damage
• Written in VBScript
• E-mail
  • Subject: “ILOVEYOU”
  • Attachment “LOVE-LETTER-FOR-YOU.TXT.vbs”
• Overwrote important files with a copy of itself
• Sent out itself to everyone in a user’s contact list
Computer Worm

- Replicates over the network (usually by itself)
  - First worm appeared at Xerox PARC in 1978

- What a worm can do?
  - Replicates itself, and thus consumes network bandwidth
  - Deletes files on a host system
  - Sends documents via e-mail
  - Carries other executables as a payload
    - Installs a backdoor in an infected computer (zombie computer)

- Modern worms
  - Large scale infection
  - Fast spread rate
    - spread over the Internet within a second
Timeline of Notable Worms (1)

- Nov 1988, Morris worm
  - First well-known worm
- March 1999, Melissa (E-mail worm)
  - Targeting Microsoft Word & Outlook-based systems
- May 2000, VBS/Loveletter or ILOVEYOU (E-mail worm)
  - Caused an estimated $10 billion in economic damage
- July 2001, Code Red (Exploited IIS bugs)
  - Considerably slowed down Internet traffic
- Jan 2003, SQL Slammer (Exploited MS SQL Server bugs)
  - Very fast: infected most of its 75,000 victims within ten minutes
  - Amazingly small, only 376 bytes
Timeline of Notable Worms (2)

• Aug 2003, Blaster, Welchia (Nachi), SoBig
  • **Blaster** (Exploited DCOM RPC bugs)
    • Coded to start a SYN flood on Aug 15 against windowsupdate.com
  • **Welchia (Nachi)**
    • A goodwill worm to remove Blaster and patch Windows
  • **SoBig** (E-mail worm)
    • Infected millions of Windows computers in Aug 2003
    • Microsoft wanted information of the worm creator for $250,000

• Apr 2004, **Sasser** (Exploited LSASSS bugs)
  • Affected:

• Jan 2007, **Storm worm**
  • Very stealthy, established botnets.
  • Used obfuscation and rootkit-techniques to hide its behavior as well as its presence
Code Red

• Released on July 13, 2001
• Considerably slowed down the Internet traffic
• Details:
  • Attacked computers running Microsoft’s IIS web server
  • Defaced the affected web site
  • Tried to spread itself by looking for more IIS servers on the Internet
  • Waited 20-27 days after it was installed to launch DoS attacks on several fixed IP addresses, including White House.
• Exploited a buffer overflow vulnerability in IIS; Used illegal GET requests to trigger the vulnerability
SLAMMER

- January 2003
- Caused DoS on some Internet hosts and dramatically slowed down general Internet traffic
- Fast
  - Infect most of its 75,000 victims within ten minutes
- A buffer overflow based attack targeting Microsoft SQL Server
- Amazingly small, only 376 bytes
- Generate random IP addresses and send itself out to those addresses.
- If the selected address happens to belong to a host that is running an unpatched copy of Microsoft SQL Server, the host immediately becomes infected and begin spraying the Internet with more copies of the worm program.
- Only stays in memory.
Blaster

• Spread during August 2003 (first noticed on August 11, peaked on August 13)
• Programmed to start a SYN flood on August 15 against port 80 of windowsupdate.com.
• Exploited a buffer overflow in the DCOM RPC service on the affected Windows operating systems
Welchia (Nachi)

- Welchia (Nachi), a worm that tries to remove the Blaster worm and patch Windows
  - Discovered in August 18, 2003

- Not good
  - Create vast amount of network traffic, thereby slowing down the Internet
  - Make the system unstable (e.g. reboot after patching)
  - Without user’s explicit consent
SoBig

• Consequences:
  • Infected millions of Microsoft Windows computers in August 2003
  • Microsoft wanted information of the worm creator for $250,000

• Details:
  • Appear as an e-mail with one of the following subjects:
    • Re: Approved       Re: Details       Re: Thank you       ...
  • Contain the text: “See the attached file for details” or the like
  • Contain an attachment by one of the following names:
    • application.pif    details.pif    thank_you.pif    ...

• Infection and spreading
  • Infect a host computer once the attachment is opened
  • Replicate by sending out the above-mentioned emails
  • E-mail addresses are gathered from files on the host computer
MyDoom

• First sighted on January 26, 2004.
• One of the fastest spreading e-mail worms

Details
• Primarily transmitted via e-mail, appearing as a transmission error
• Subject lines including “Error”, “Mail Delivery System”, “Test” or “Mail Transaction Failed”
• Contains a malicious attachment

Infection and Spreading
• Resend the worm to e-mail addresses found in local files once the attachment is opened.
• Copies itself to the “shared folder” of KaZaA (a P2P file-sharing app)

Backdoor
• Installs a backdoor on port 3127/tcp to allow remote control of the subverted PC
• A DoS attack against SCO Group, Microsoft, and antivirus sites
Sasser

• First noticed in April 2004. Affected:

  - Scan different ranges of IP addresses and connect to victims’ computers primarily through TCP port 445.

• Can be easily stopped by a properly configured firewall, or by downloading patches

• Can spread without the help of the user.
  - Exploit a buffer overflow in LSASS (Local Security Authority Subsystem Service)
  - Scan different ranges of IP addresses and connect to victims’ computers primarily through TCP port 445.
Goals of Worms

- “bragging rights” in early days
  - infect as many sites as possible
  - be as noticeable as possible
  - values fast spread, DoS effect

- Detection techniques could hence be targeted at these features

- More recently, worms used to establish botnets
  - Need to remain stealthy
    - Spread slowly so as to evade detection
    - Attacks launched on demand, but infection itself should not cause any noticeable surge in network traffic or other feature changes that can be easily spotted
    - So, we no longer hear about “high-profile” worms.
Distributed Denial-of-Service (DDoS)

- **DoS**
  - An attack on a computer system or network that causes a loss of service to users

- **Methods**
  - Consumption of computational resources, such as bandwidth, disk space, or CPU time
  - Disruption of configuration information, such as routing information
  - Disruption of physical network components

- **DDoS**
  - Use of multiple hosts (often through Botnet) in a DoS
What is a Botnet?
- A collection of compromised computers
- The computers are implanted with backdoor programs
  - Usually by worms, viruses
- The programs are under a common control infrastructure
- Botnet’s originator can control the group remotely
  - Earlier botnets used means such as IRC
  - But modern botnets have begun to rely means that are harder to spot
    - HTTP
    - P2P networks

Purpose
- DDoS
- SMTP mail relays for SPAM
- Theft of sensitive information
  - E.g. login IDs, credit card numbers, application serial numbers
Rootkit

• Stealthy backdoor programs

• Intended to maintain “invisibility” of intruders
  • Intercepts data from terminals, network connections, and the keyboard
  • Conceals logins, running processes, files, logs, or other system data

• Origins of “rootkit”
  • Originally referred to such kind of programs in Unix systems (root – the administrator)
Rootkits

• Userlevel rootkits
  • Early ones on UNIX used to replace many programs used to examine system state
    • `ls`, `ps`, `netstat`,…
  • Drawback: if an administrator uses a custom C-program to examine system state, he can discover the presence of rootkit

• Kernel rootkits
  • System call interception based
    • All user level requests are intercepted and modified to hide the presence of rootkit
    • Problem: can be difficult to block all ways to learning about the presence of rootkit
More Advanced Rootkits

• May reside entirely within the kernel, with no user-level processes
• Hide themselves from system monitoring tools
  • e.g., put themselves on a scheduler queue, but not task queue
• In the most extreme case, avoid changing any data that is predictable or is read-only
  • Hide within kernel data structures that change all the time
• Rootkits that hide underneath the OS
  • Lift the OS into a VM!
SonyBMG DRM Rootkit (2005)

• Extended Copy Protection (XCP) DRM for CD copy protection
  • User is required to install XCP software contained in the CD to play XCP-protected CD on a Windows system.
  • XCP intercepts all accesses of the CD drive and only allows XCP-bundled media player to access music tracks on the CD
  • (Rootkit) XCP conceals itself from the user by installing a patch to the Windows operating system. This patch stops ordinary system tools from displaying processes, registry entries, or files who names begin with $sys$.

• About 4.7 million XCP-CDs shipped, 2.1 million sold [New York Times]
SonyBMG DRM Rootkit (2005)

- A Controversial DRM mechanism

- Weakened system security
  - XCP rootkit could be used by other malware
    - The first one was discovered in November 2005
  - XCP uninstaller, which was released later, left serious security holes on the system

- This episode set back DRM efforts for a while
  - But ultimately, DRM forces won
    - Included in HTML 5 (March 2017)
    - Relies on protection offered by the platform (iOS, Adroid, Kindle)
    - or code obfuscation (e.g., SilverLight, Flash)
      - Obfuscation is security by obscurity, but pretty effective in practice because the effort (for defeating) is not often worth the cost.
      - Piracy is a more serious problem where cost is relatively high
Microsoft to Zap Sony DRM 'Rootkit'

By Ryan Naraine
November 12, 2005

Microsoft Corp. will start deleting the rootkit component of the controversial DRM scheme used by Sony BMG Music Entertainment.

The software giant's Windows AntiSpyware application will be updated to add a detection and removal signature for the rootkit features used in the XCP digital rights management technology.
Spyware

• **Properties**
  • Intercept or take partial control of computer’s operation
  • Without the informed consent of that computer’s legitimate user.
  • Does not usually self-replicate.

• **Purpose**
  • Delivery of unsolicited pop-up advertisements
  • Theft of personal information
  • Monitoring of Web-browsing activity for marketing purposes
  • Routing of HTTP request to advertising sites
Spam

• Properties
  • Sending of *unsolicited* (commercial) emails
  • Sending nearly identical messages to thousands (or millions) of recipients

• Spamming in different media
  • E-mail spam, Messaging spam, Newsgroup spam and Forum spam, Mobile phone spam, Internet telephony spam, Blog, wiki, guestbook, and referrer spam, etc
Spam

- Spam volumes hold steady
  - After falling from a high of 6T to about 1T/month
- Expands to social networks
  - Facebook, Twitter, Instagram, ...

Source: McAfee Threats Report: First Quarter 2013
Phishing

- Uses social engineering techniques
  - Masquerading as a trustworthy person or business in an apparently official electronic communication
  - Attempts to fraudulently acquire sensitive information
    - Such as passwords and credit card details

- Spear-phishing
  - Phishing attack that is narrowly targeted at a single individual or a group of individuals
Dear SouthTrust bank customer,

Technical services of the SouthTrust bank are carrying out a planned software upgrade. We earnestly ask you to visit the following link to start the procedure of confirmation of customers' data.

https://www.southtrust.com/st/PersonalBanking/custdetailsconfirmation

Please do not answer to this email – follow the instructions given above.

We present our apologies and thank you for co-operating.
Phishing

- New types of phishing
  - Watering hole
  - Clone phishing
  - Tabnapping

Source: Internet Security Threat Report 2013, Symantec

Source: McAfee Threats Report: First Quarter 2013
Online DDoS Extortion

- Extortion: you pay us or you will be attacked
- [CMU and Information Week, 2004]
  - 17% of companies surveyed are victims of online extortion.
- [Alan Paller, SANS Institute, 2004]
  - 6 or 7 thousand organizations are paying extortion
  - Every online gambling site is paying extortion
- Currently, targets seem to be more selected
  - “Shady” businesses, e.g., Online gambling
Botnets & DDOS

• Botnets now include mobile devices
  • Android botnets
    • [http://mobile.slashdot.org/story/13/01/19/0735259/android-botnet-infects-1-million-plus-phones](http://mobile.slashdot.org/story/13/01/19/0735259/android-botnet-infects-1-million-plus-phones)

• DDoS used as a diversion
Web Vulnerabilities

Scanned Websites with Vulnerabilities

A critical vulnerability is one which, if exploited, may allow malicious code to be run without user interaction, potentially resulting in a data breach and further compromise of visitors to the affected websites.

<table>
<thead>
<tr>
<th></th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerabilities</td>
<td>77%</td>
<td>76%</td>
<td>78%</td>
</tr>
<tr>
<td>Change</td>
<td>-1% pts</td>
<td>+2% pts</td>
<td></td>
</tr>
</tbody>
</table>

Percentage of Vulnerabilities Which Were Critical

<table>
<thead>
<tr>
<th></th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>16%</td>
<td>20%</td>
<td>15%</td>
</tr>
<tr>
<td>Change</td>
<td>+4% pts</td>
<td></td>
<td>-5% pts</td>
</tr>
</tbody>
</table>
Zero Day Exploits ...
Ransomware ...
Data Leaks ...

- **Anthem**: 80,000,000
- **Experian / T-mobile**: Hacking Team
- **IRIS Invest Bank**: Kromtech
- **Premera**: Carefirst
- **AshleyMadison.com**: Carphone Warehouse
- **Honda**: British Airways
- **Home Depot**: 56,000,000
- **Securus Technologies**: 70 million
- **MSpy**: Mozilla
- **Voter Database**: 191 million
- **Sanrio**: Staples
- **US Office of Personnel Management**: TalkTalk
- **VTech**: US Office of Personnel Management (2nd Breach)
- **AOL**: 2,400,000
- **Ebay**: 145,000,000
- **JP Morgan Chase**: 76,000,000
- **Korea Credit Bureau**: Neiman Marcus
- **Target**: 70,000,000
- **LinkedIn**: Japanese Airlines
- **Adobe**: 36,000,000
- **Evernote**: 50,000,000
- **Living Social**: 50,000,000
- **LivingSocial**: NMBS
- **Massive American business**: OVH
- **UBISOFT**: "unknown"
- **Zappos**: 24,000,000
Underlying Causes

- Untrusted software
  - Malware, including viruses, worms, bots, ...
- Configuration errors
  - Default passwords, permissive firewall rules, ...
- Human element
  - Insider threats, operator mistakes, social engineering
- Vulnerabilities in trusted software
  - These may be the result of errors in
    - Threat modeling
    - Design/logic
    - Implementation
    - Testing
Stealth and Obfuscation

• Malware wants to remain stealthy
  • So that it can be used in cyber crime (or to achieve other goals of attacker) without being detected
  • Protect “intellectual property”

• Intellectual property protection for legitimate code
  • Make it difficult to reverse-engineer code
  • Introduce watermarks
  • Prevent unauthorized copy of content

• Result
  • Obfuscation techniques
Types of obfuscation

• To thwart static analysis (code obfuscation):
  • Low-level code obfuscations
    ▪ Insert data in the middle of code
    ▪ Violate typical ABI conventions, e.g., call/return, stack use, jumping to the middle of code, dynamic generation or modification code, etc.
  • Code encryption and transformation
  • Higher level code obfuscation
    ▪ Rename functions and variables
    ▪ Control-flow obfuscation
  • Data obfuscation

• To thwart dynamic analysis (behavior obfuscation):
  • Evasion: Carefully match behavior with that of benign software, or employ code/behaviors that do not trigger suspicion
  • Anti-analysis techniques
    ▪ Detect execution within VM, emulator, or a sandbox and alter behavior
  • Combine benign and malicious behaviors, complicating detection
Polymorphic viruses and encryption

- Historically, virus detection relied on “signatures” that captured byte sequences in code that were unique to the virus

- Polymorphism
  - Encrypt virus code so that it can change from one instance to another
  - Basically, change the encryption key from one generation to the next, causing massive changes to byte sequences

- Defense
  - Focus on invariant parts used to pack/unpack
  - Capture unpack/launch behavior (runtime detection)
  - Run virus scanner after unpack
Obfuscation/Metamorphic Viruses

- Metamorphic viruses rewrite their entire code from one generation to next
- No “fixed” part in their code
  - Need not have any code encryption/decryption, so behavior based techniques can be defeated as well
- Metamorphic techniques
  - Use alternative instruction sequences to achieve the same effect
  - More general program obfuscation techniques
Control-flow Obfuscation

- Split or aggregate
  - Basic blocks
  - Loops
    - e.g., one loop becomes two loops or vice-versa
  - Procedures
    - Replace one procedure by two or merge two procedures
    - Inline a procedure, or outline (i.e., create new procedure)

- Reorder

- Insert dead-code (i.e., unreachable code)
  - Obfuscate using conditions

- Replace instruction sequences w/ alternate ones
- Insert conditional jumps using “opaque” predicates
- Insert indirect jumps
- Exploit aliasing and memory errors
Data Obfuscation

- Rename variables
- Split or aggregate variables
  - Split structures into individual variables or vice-versa
- Split individual variables
  - E.g., A = B - C – instead of A, use B and C
  - Clone a variable
- Pad arrays (and possibly structures) with junk elements
- “Encrypt” data values
- Introduce extra levels of indirection
  - Instead of a simple variable, declare a pointer
- Introduce aliasing
- Introduce memory errors
- Introduce additional (or remove) function parameters
Key Issues in Malware Defense

• Plenty of motivation for attackers to remain stealthy
  • Many techniques are available to achieve this
    • Anti-virtualization, anti-analysis, obfuscation, ...

• Adaptive
  • Will employ evasion techniques specifically designed to defeat commonly deployed defenses

• Need to assume a strong adversary model
  • Rely on self-protecting defense techniques
    • Ensure defense mechanisms are not compromised by malware
  • Complete mediation
  • Robustness against multi-step attacks (“stepping stones”)
The Human Element

• Growing system complexity contributes to more operator errors
  • misconfigured systems
  • especially problematic in settings where many components interact

• Insider attacks

• Social engineering attacks

• Intentionally introduced vulnerabilities
  • Infiltration into key software or open-source teams
  • Hardware Trojans
Securing Untrusted Code
Untrusted Code

◆ May be untrustworthy
  ▪ Intended to be benign, but may be full of vulnerabilities
  ▪ These vulnerabilities may be exploited by attackers (or other malicious processes) to run malicious code

◆ Or, may directly be malicious: may use
  ▪ Obfuscation
    ▼ Code obfuscation
    ▼ Anti-analysis techniques
    ▼ Use of vulnerabilities to hide behavior
  ▪ (Behavioral) evasion
    ▼ Actively subvert enforcement mechanisms

◆ Security is still defined in terms of policies
  ▪ But enforcement mechanisms need to be stronger in order to defeat a strong adversary.
Reference Monitors

- Security policies can be enforced by reference monitors (RM)
  - Key requirements
    - Complete mediation
    - (If interaction with user is needed) Trusted path

- With benign code, we typically assume that it won’t actively evade enforcement mechanisms
  - We can possibly maintain security even if there are ways to subvert the checks made by the RM
Types of Reference Monitors

- **External RM**
  - RM resides outside the address space of untrusted process
  - Relies on memory protection
    - Protect RM’s data from untrusted code
    - Limit access to RM’s code

- **Inline RM**
  - Policy enforcement code runs within the address space of the untrusted process
  - Cannot rely on traditional hardware-based memory protection
External Reference Monitors

- System-call based RMs
- Linux Security Modules (LSM)
- AppArmor
System-call based RM

- OSes already implement RM to enforce OS security policies
  - Most aspects of policy are configured (e.g., file permissions), while the RM mainly includes mechanisms to enforce these policies
- But these are typically not flexible enough or customizable
- More powerful and flexible policies may be realized using a customized RM
- System-calls provide a natural interface at which such a customized RM can reside and mediate requests.
Why monitor system calls?

- Complete mediation: All security-relevant actions of processes are administered through this interface.
- Performance: Associated with a context-switch --- can be exploited to protect RM without extra overheads.
- Granularity
  - Finer granularity than typical access control primitives.
  - But coarse enough to be tractable: a few hundred system calls.
- Expressiveness
  - Clearly defined, semantically meaningful, well-understood and well-documented interface (except for some OSes like Windows).
  - Orthogonal (each system call provides a function that is independent of other system calls --- functions that rarely, if ever, overlap).
  - Can control operations for which OS access controls are ineffective, e.g., loading modules.
    - A large number of security-critical operations are traditionally lumped into “administrative privilege”.
- Portability: System call policies can be easily ported across similar OSes, e.g., various flavors of UNIX.
Some drawbacks of system calls

- **Interface is designed for functionality**
  - Several syscalls may be equivalent for security purposes, but we a syscall policy needs to treat them separately

- **Not all relevant operations are visible**
  - For instance, syscall policies cannot control name-to-file translations

- **Race conditions**
  - Pathname based policies are prone to race conditions
  - More generally, there may be TOCTTOU races relating to system call arguments
    - Unless the argument data is first copied into RM, checked, and then this checked copy is used by the system call
      - Adds more complexity
  - The window for exploiting TOCTTOU attacks can be increased by using a large sequence of symbolic links in the name
Linux Security Module Framework

- Motivated by the drawbacks of syscall monitors
- Defines a number of “hooks” within Linux kernel
  - Includes all points where security checks need to be done
  - RMs can register to be invoked at these hooks
  - SELinux, as well as Linux capabilities are implemented using such RMs
- Drawbacks
  - The framework has significant complexity --- while it simplifies some things, the increased complexity makes other things hard.
  - Requires a lot of effort to identify the things that need checking, and where all the hooks need to be placed
  - Very closely tied to the implementation details of an OS --- not easily ported to other OSes.
System call interposition approaches

**User-level interception**
- RM resides within a process
  - Library interposition
    - RM resides in the same address space
    - Advantages
      - high performance
      - Potential for intercepting higher level (semantically richer) operations
    - Drawbacks: RM is unprotected, so appropriate only for benign code
  - Kernel-supported interposition, with RM residing in another process
    - Advantages: Secure for untrusted code
    - Drawback: High overheads due to context switches
    - Example: ptrace interface on Linux

**Kernel interception**
- The RM resides in the kernel
- Advantages: high performance, secure for untrusted code
- Drawbacks:
  - difficult to program
  - requires root privilege
  - Rootkit defense measures pose compatibility issues
Inline Reference Monitoring

- Foundations
  - Software Fault Isolation (SFI)
  - Control-flow Integrity (CFI)
- Case Study
  - Google Native Client (NaCl)
Inline Reference Monitors (IRM$s$)

- **Provide finer granularity**
  - “Variable $x$ is always greater than $y$”
  - Provides much more expressive power

- **Very efficient**
  - Does not require a context switch

- **Key challenge:**
  - Protecting IRM from hostile code
Securing RM in the same address space

- **Protect RM data that is used in enforcing policy**
  - Software-based fault isolation (SFI)

- **Protect RM checks from being bypassed**
  - Control-flow integrity (CFI)

**Note**
- For vulnerability defenses (e.g., Stackguard), we implement the checks using an IRM.
- But we don’t worry so much about these properties since we are dealing with benign (and not malicious) code.
Software Fault Isolation (SFI)
Background

 Fault Isolation

- What is fault isolation?
  - when "something bad" happens, the negative consequences are limited in scope.
- Why is it needed?
  - Untrusted plug-ins makes applications unreliable
  - Third-party modules make the OS unreliable

 Hardware based Fault Isolation

- Isolated Address Space
- RPC interfaces for cross boundary communication
SFI [Wahbe et al 1994]

 Emblem: Motivation
 - Hardware-assisted context-switches are expensive
   - TLB flushing; some caches may require flushing as well

 Emblem: Key idea
 - Insert inline checks to verify memory address bounds for
   - Data accesses
   - Indirect control-flow transfers (CFT)
     - Direct CFTs can be statically checked

 Emblem: Challenges
 - Efficiency
   - each memory access has the overhead of checking
 - Security
   - Preventing circumvention or subversion of checks
Even when running in the same virtual address space, limit some code components to access only a part of the address space. This subspace is called a “fault domain”
Software Fault Isolation

❖ Virtual address segments
  ▪ Fault domain (guest) has **two segments**, one for code, the other for data.
  ▪ Each segment share a **unique upper bits** (segment identifier)
  ▪ Untrusted module can **ONLY jump to or write** to the same upper bit pattern (segment identifier)

❖ Components of the technique
  ▪ Segment Matching
    ▼ Optimization: instead of checking, simply override the segment bits
      – Originally, the term “sandboxing” referred to this overriding
  ▪ Data sharing
  ▪ Cross-domain Communication
Segment Matching

- Insert checking code before every **unsafe instructions**
  - To prevent subversion of checks, use dedicated registers, and ensure that all jumps and stores use these registers
    - Need only worry about indirect accesses
    - Don't forget that returns are indirect jumps too
- Checking code determines whether the unsafe instruction has the correct **segment identifier**
- Trap to a system error routine if checking fails – pinpoint the offending instruction
Segment Matching

dedicated-reg ← target address
  Move target address into dedicated register.
scratch-reg ← (dedicated-reg >> shift-reg)
  Right-shift address to get segment identifier.
scratch-reg is not a dedicated register.
shift-reg is a dedicated register.
compare scratch-reg and segment-reg
  segment-reg is a dedicated register.
trap if not equal
  Trap if store address is outside of segment.
store instruction uses dedicated-reg

5 instructions, Need 5 dedicated registers (segment value needs to be different for code and data) and it can pinpoint the source of faults. Can reduce the number of registers by hard-coding some values (e.g., number of shift bits).
Optimization 1: Address Sandboxing

- Reduce runtime overhead further compared to segment matching by not pinpointing the offending instruction
- Before each unsafe instruction, inserting codes can set the upper bits of the target address to the correct segment identifier
Address Sandboxing

dedicated-reg ← target-reg\&\text{and-mask-reg}
Use dedicated register and-mask-reg
to clear segment identifier bits.
dedicated-reg ← dedicated-reg|\text{segment-reg}
Use dedicated register segment-reg
to set segment identifier bits.
store instruction uses dedicated-reg

3 instructions, Require 5 dedicated registers (since mask and segment
registers will be different for code and data)

Correctness: Relies on the \textit{invariant} that dedicated registers always contain
valid values before any control transfer instruction.
Optimization 2: Guarding pages

- A single instruction accesses multiple bytes of memory (4, 8, or may be more)
- Need to check whether all bytes are within the segment
  - Require at least two checks!
- Optimization
  - Sandboxing reg, ignore reg+offset
  - Guard zones ensure that reg+offset will also be in bounds (or that there will be a hardware fault)

Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.
Data sharing

- Read-only sharing can be achieved in several ways:
  - Option 1: Don’t restrict read accesses
  - Option 2: Allow reads to access some segments other than that of untrusted code
  - Option 3: Remap shared memory into the address space of both the untrusted and trusted domains

- Read-write sharing can use similar techniques.
cross fault domain communication

- trusted stubs to handle RPC
  - for each pair of fault domains
  - stub: copy arguments, re/store registers, switch the exe. stack, validate dedicated regs but! no traps or address space switching (thus, cheaper than HW RPC)

- jump tables to transfer control
  - consists of jump instructions of which target address is legal, outside the domain
SFI details (continued)

- **Need compiler assistance**
  - To set aside dedicated registers
  - *But we cannot trust the compiler*
    - Programs may be distributed as binaries, and we can’t trust the compiler used to compile that untrusted binary

- **Need a verifier**
  - Verification is quite simple
    - Dedicated registers should be loaded only after address-sandboxing operations
    - All direct memory accesses and direct jumps should stay within untrusted domain. Implementation operates on binary code
      - Note that SFI checks all indirect accesses and control-transfers at runtime
  - Was implemented on RISC architectures

- **Precursor to proof-carrying code [Necula et al]**
  - Code producer provides the proof, consumer needs to check it.
    - Proof-checking is much easier than proof generation
    - Especially in an automated verification setting:
      - producer needs to navigate a humongous search space to construct a proof tree
      - consumer needs to just verify that the particular tree provided is valid
Difficulties of bringing SFI to CISC

- Problem 1: Variable-length instructions
  - What happens if code jumps to the middle of an instruction

- Problem 2: Insufficient registers
  - SFI requires 5 dedicated registers for segment matching
  - SFI requires 5 dedicated registers for address sandboxing
  - x86 has very few general-purpose registers available
    - eax, ebx, ecx, edx, esi, edi
  - PittsSFIeld: uses ebx as a dedicated register AND treats esp and ebp as sandboxed registers (adds needed checks)
Solution to Problem 1

- **padding with no-ops to enforce alignment constraints (power of two)**
  - because CISC architectures allow various instruction streams, which makes SFI harder

- **call placed at the end of chunks**
  - because the next addresses are targets of returns
  - they also have **low 4 bits zero** due to 16 bytes align

- **put unsafe operation and its corresponding check together in a chunk**
  - atomic, i.e. unsafe op. must be followed by check; no dedicated registers required
Solution to Problem 2

- **Hardcode segments**
  - Avoids need for segment registers etc.

- **Make code and data segments adjacent, and differ by only one bit in their addresses**
  - Sandboxing now achieved using a single instruction
    - and 0x20ffffff, %ebx
    - Store using ebx
  - For indirect jumps, use:
    - and 0x10fffffff, %ebx
    - Jump using ebx

- **Alternative approach**
  - Use x86 segment (CS, DS, ES) registers!
    - Very efficient but not available on x86_64
Control Flow Integrity (CFI)
Control-flow Integrity (CFI) [Abadi et al]

- Unrestricted control-flow transfers (CFTs) can subvert the IRM
  - Simply jump past checks, or
  - Jump into IRM code that updates critical IRM data

- Approaches
  - Compute a control-flow graph using static analysis, enforce it at runtime
    - Benefits: With accurate static analysis, can closely constrain CFTs.
    - Drawback: Requires reasoning about targets of indirect CFTs (hard!)
  - Enforce coarse-grained CFI properties
    - All calls should go to beginning of functions
    - All returns should go to instructions following calls
    - No control flow transfers can target instructions belonging to IRM
CFI (Continued)

- **Coarse-grained version is sufficient to protect IRM**
  - Like SFI, CFI is self-protecting
    - CFI checks the targets of jump, so it can prevent unsafe CFTs that attempt to jump just beyond CFI checks
    - In PittSFIeld, this was achieved by ensuring that the check and access operations were within the same bundle
      - Jumps can only go to the beginning of a bundle, so you can't jump between check and use
  - Because of this, SFI and CFI provide a foundation for securing untrusted code using inline checks.
  - CFI can also be applied to protect against control-flow hijack attacks
    - Jump to injected code (easy)
    - Return to libc (most obvious cases are easy)
    - Return-oriented programming (requires considerable effort to devise ROP attacks that defeat CFI)
    - **But not a foolproof defense**

- **In addition:**
  - IRM code shouldn’t assume that untrusted code will follow ABI conventions on register use
  - IRM code should use a separate stack
    - To prevent return-to-libc style attacks within IRM code
CFI Implementation Strategies

◆ Approach 1 (proposed in the original CFI paper)
  - Associate a constant index with each CFT target
  - Verify this index before each CFT
    ▼ Ideal for fine-grained approach, where static analysis has computed all potential targets of each indirect CFT instruction
  - Issues
    ▼ If locations L1 and L2 can be targets of an indirect CFT, then both locations should be given the same index
    ▼ If another CFT can go to either L2 or L3, then all three must have same index
    ▼ A particular problem when you consider returns
      – Accuracy can be improved by using a stack, but then you run into the same compatibility issues as stacksmashing defenses that store a second copy of return address
## CFI Instrumentation

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>mov eax, [esp+4]</td>
</tr>
<tr>
<td></td>
<td>; computed jump</td>
<td>; dst</td>
</tr>
<tr>
<td>8B 44 24 04</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Method (a): unsafe, since ID is *embedded* in callsite (could be used by attacker)

Method (b): safe, but pollute the data cache

---

**Figure 2:** Example CFI instrumentations of a source x86 instruction and one of its destinations.
CFI Implementation

- **CFG construction is conservative**
  - Each computed call instruction may go to **ANY** function whose address is taken (**too coarse**)
  - Discover those functions by checking relocation entries.
    - Won’t work on stripped code
CFI Assumption

**UNQ**: Unique IDs.
- choose longer ID to prevent ensuring uniqueness
- Otherwise: jump in the middle of a instruction or arbitrary place (in data or code)

**NWC**: Non-Writable Code.
- Code could not be modified. Otherwise, verifier is meaningless, thus all the work is meaningless......

**NXD**: Non-Executable Data
- Otherwise, attacker can execute data that begins with a correct ID.

All the assumptions should hold. Otherwise, this CFI implementation can be defeated.
CFI Implementation Strategies

- **Approach 2**
  - Use an array $V$ indexed by address, and holding the following values
    - `Function_begin`, `Valid_return`, `Valid_target`, `Invalid`
  - A call to target $X$ is permitted if $V[X] == Function_begin$
  - A return to target $X$ is permitted if $V[X] == Valid_return$
  - A jump to target $X$ is permitted if $V[X] != Invalid$
  - Otherwise, CFT is not permitted

Note that CFI implementations need only check indirect CFTs
SFI, CFI and Follow-ups

- SFI originally implemented for RISC instruction set, later extended to x86
  - Efficient implementation on x86, x86-64 and ARM architectures have been the focus of recent works

- CFI originally implemented using Microsoft’s Phoenix compiler framework
  - Binary instrumentation requires a lot of information unavailable in normal binaries, and hence reliance on specific compiler
  - But the concept has had broad impact

- Google’s Native Client (NaCl) project is the most visible application of SFI and CFI techniques
  - Supports untrusted native code in browsers
  - Part of recent WebAssembly standard
  - Included in Firefox 52 and later
Case Study:
Google Native Client (NaCl)
Motivation

- Browsers already allow Javascript code from arbitrary sites, but its performance is inadequate for some applications
  - Games
  - Fluid dynamics (physics simulation)
- Permitting native code from arbitrary sites is too dangerous!
Native Client Approach

- **Sandboxed environment for execution of native code. Two parts:**
  - SFI using x86 segment as inner sandbox
  - Runtime for allowing safe operations from outer sandbox

- **Good runtime facilities**
  - Multi-threading support
  - IPC: PPAPI
  - Performance: 5% overhead on average
System Architecture

Browser Process

- JavaScript
  - untrusted
- PPAPI

Service Runtime

- Guest data
- Guest Code
  - Native Client Module
  - Native Client Plug-in
  - Native Client Process
  - IMC
Design

◆ Inner Sandbox
  ▪ Static verification to ensure all security properties hold for the untrusted code
  ▪ 32-byte instruction bundles to ensure CFI
  ▪ Trampoline/springboard to allow safe control transfer from untrusted to trusted and vice versa

◆ Runtime Facilities
  ▪ Safe execution of possible “unsafe” operations
  ▪ Inter module communication: PPAPI & IMC
Binary Constraints & Properties

**Constraints**

- No self modifying code
- Static linked with a fix start address of text segment
- All indirect control transfer use `nacljmp` instruction
- The binary is padded up to the nearest page with `hlt`
- No instructions overlap 32-byte boundary
- All instructions are reachable by fall-through disassembly from starting address
- All direct control transfers target valid instructions
Control Flow Integrity

- All control transfers must target an instruction identified during disassembly

- Direct control flow
  - Target should be one of reachable instructions

- Indirect Control flow
  - Segmented support (works because a fix start address)
  - No returns
  - Limit target to 32 byte boundary (*nacljmp on the right*)
    \[
    \text{jmp eax} \rightarrow \text{and eax,0xffffffffe0}
    \]
    \[
    \text{jmp eax}
    \]
  - Nacljmp is atomic
Data Integrity

- Segmented memory support

- Limited instruction set (no assignment to segment register)
  - i.e. move ds, ax is forbidden
Web Security
Historical Web

• Historically, the web was just a request response protocol
• HTTP is stateless, which means that the server essentially processes a request independent of prior history
• Envisioned as a way for exchanging information
Current Web

• Evolving into a platform for executing programs that support day-to-day tasks
• A lot of state needs to be maintained
• Distributed computation, and trust model
HTTP Requests

• A request has the form:

```
<METHOD> /path/to/resource?query_string HTTP/1.1
<header>*
<BODY>
```

• HTTP supports a variety of methods, but only two matter in practice:
  – GET: intended for information retrieval
    • Typically the BODY is empty
  – POST: intended for submitting information
    • Typically the BODY contains the submitted information
Structure of HTTP GET request

- Connect to: www.example.com
  - TCP Port 80 is the default for http, others may be specified explicitly in the URL.
- Send: GET /index.html HTTP/1.1
- Server Response:
  
  HTTP/1.1 200 OK  
  Date: Mon, 23 May 2005 22:38:34 GMT  
  Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)  
  Etag: "3f80f-1b6-3e1cb03b"  
  Content-Length: 438  
  Connection: close  
  Content-Type: text/html; charset=UTF-8
GET with parameters

• GET /submit_order?sessionid=79adjadf888888768&pay=yes
  HTTP/1.1

• User inputs sent as parameters to the request
POST Requests

- Another way of sending requests to HTTP servers
- Commonly used in FORM submissions
- Message written in the BODY of the request
- Sending links with malicious parameter values is difficult when a web site accepts only POST requests.
- But a script running on a malicious web site can as easily send a POST request (as a GET request) to another web site.
HTTP Responses

• A response has the form

```plaintext
HTTP/1.1 <STATUS CODE> <STATUS MESSAGE>
<header>*
</header>

<BODY>
```

• Important response codes:
  – 2XX: Success, e.g. 200 OK
  – 3XX: Redirection, e.g. 301 Moved Permanently
  – 4XX: Client side error, e.g. 404 Not Found
  – 5XX: Server side error, e.g. 500 Internal Server Error

HTTP response

HTTP/1.1 200 OK
Date: Tue, 21 Oct 2014 16:21:44 GMT
Server: Apache/2.2.25 (Unix) mod_ssl/2.2.25 OpenSSL/1.0.1h PHP/5.2.17
Last-Modified: Tue, 21 Oct 2014 15:37:09 GMT
ETag: "3aaa5c-850-505f09ab7f211"
Accept-Ranges: bytes
Content-Length: 2128
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>

<title>Is The Internet On Fire?</title>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
<link rev="made" href="mailto:jschauma@netmeister.org">

</head>
Cookies

- HTTP is stateless, therefore client needs to remember state and send this with every request
- Cookies are the common way of keeping state
  - Client:
    GET /index.html HTTP/1.1
  - Server:
    HTTP/1.1 200 OK
    Content-type: text/html
    Set-Cookie: sess-id=3773777adbdad

(content of page)
Cookies...

• Browsers send cookie with every subsequent request
  –GET /spec.html HTTP/1.1
    Host: www.example.org
    Cookie: sess-id=3773777adbdad

• Now server can look up stored state through sess-id

• Alternative to cookies: hidden form fields.
What Are Cookies Used For?

• Authentication
  – The cookie proves to the website that the client previously authenticated correctly

• Personalization
  – Helps the website recognize the user from a previous visit

• Tracking
  – Follow the user from site to site; learn his/her browsing behavior, preferences, and so on
Sessions

• As long as different users have different session identifiers (present in their cookies), the web server will be able to tell them apart
  – Regardless of their IP address

• When users delete their cookies, the browsers no longer send out the appropriate session identifier, and thus the web server “forgets” about them
Session Identifiers

• Long pseudo-random strings
• Unique per visiting client
• Each identifier is associated with a specific visitor
  – ID A -> User A
• As sensitive as credentials (per session)
One missing piece

- We can create websites
- And we can have state, enabling us to have a personalized web
  - Banking, Email, Social networks, etc.
- But our pages are still static
  - The server sent some HTML, the browser drew it on the screen, and that’s it
JavaScript

• “The world’s most misunderstood programming language”

• Language executed by the Web browser
  – Scripts are embedded in webpages
  – Can run before HTML is loaded, before page is viewed, while it is being viewed, or when leaving the page

• Used to implement “active” webpages and Web applications

• A potentially malicious webpage gets to execute some code on user’s machine
JavaScript History

• Developed by Brendan Eich at Netscape
  – Scripting language for Navigator 2
• Later standardized for browser compatibility
  – ECMAScript Edition 3 (aka JavaScript 1.5)
• Related to Java in name only
  – Name was part of a marketing deal
  – “Java is to JavaScript as car is to carpet”
• Various implementations available
  – SpiderMonkey, RhinoJava, others
Aside: Java Security

• With binary code, memory and type safety issues complicate the problem of untrusted code
• Java and Javascript rely on safe languages
  • avoid low-level issues arising in C, C++ and binary code
    • No buffer overflows.
  • code can be created and executed only through sanctioned pathways, e.g., class loader
  • access-control restrictions associated with classes will be strictly and fully enforced
    • Can’t circumvent public/private restrictions by casting etc.
Java Vs JavaScript

• Java originally developed to support “active web pages”
  • Applets were intended to allow local execution of untrusted code
  • Security was achieved by restricting access to local resources, e.g., files

• Drawbacks
  • did not provide good integration with the browser environment
  • focus was more on (OS) integrity rather than confidentiality
  • these factors led to the development of Javascript
  • Today, Adobe flash is closer in many ways to Java than Javascript
Java Vs JavaScript

• Javascript takes a different approach
  • Language safety is still the basis
  • Use this basis to provide safe interface to the browser environment
  • The security model is object-oriented
  • What are the browser resources, which ones are accessible to untrusted code
• Browser is the platform, not the underlying OS
• It is not about whether untrusted code can access local files, but whether the browser permits it to do so (“trusted dialogs”)
• Cookie-based model of browser security evolved in conjunction with Javascript, leading to excellent support for the same.
Common Uses of JavaScript

• Page embellishments and special effects
• Dynamic content manipulation
• Form validation
• Navigation systems
• Hundreds of applications
  – Google Docs, Google Maps, dashboard widgets in Mac OS X, Philips universal remotes ...
JavaScript in Webpages

• Embedded in HTML as a `<script>` element
  – Written directly inside a `<script>` element
    • `<script>` alert("Hello World!") </script>
  – In a file linked as `src` attribute of a `<script>` element
    `<script type="text/JavaScript" src="functions.js"></script>`

• Event handler attribute
  `<a href="http://www.yahoo.com"
  onmouseover="alert('hi');">`

• Pseudo-URL referenced by a link
  `<a href="javascript: alert('You clicked');">Click me</a>"
Document Object Model (DOM)

• HTML page is structured data
• DOM is object-oriented representation of the hierarchical HTML structure
  – Properties: `document.alinkColor`, `document.URL`, `document.forms[]`, `document.links[]`, ...
  – Methods: `document.write(document.referrer)`
    • These change the content of the page!
• Also Browser Object Model (BOM)
  – Window, Document, Frames[], History, Location, Navigator (type and version of browser)
Browser and Document Structure

- navigator object
- window object
- frame object
- frame object
- document object
- <p> paragraph object
- <h2> heading object

The Document Object Model:

A Web page containing JavaScript processing to the XHTML elements on that page typically have considered XHTML tags simply as markup codes providing structure to page content and supplying mechanisms through which styling is applied to that content. Importantly, though, XHTML tags are also software objects. That is, all XHTML tags have properties and methods that can be programmed. As is the case with all software objects, properties refer to characteristics of the element; methods refer to actions the object can perform. XHTML tags, then, are programmable through JavaScript processing routines that set their properties and activate their methods in order to make Web pages dynamic.

The programming interface to the XHTML object comprising Web page is known as the Document Object Model (DOM). The DOM comprises a Web page, and it provides the means for identifying their properties and methods to produce dynamic changes.

The DOM Hierarchy:

Basically, the DOM is a hierarchy of browser components. At the top-most level is the browser (navigator) object. At the next level down the hierarchy is the window object, the main browser window within which Web pages appear. Within the window are optional frame objects (if the window is divided into frames), and these window and frame objects contain the document objects representing Web pages. The page itself contains other objects, including...
Reading Properties with JavaScript

Sample script

1. document.getElementById('t1').nodeName
2. document.getElementById('t1').nodeValue
3. document.getElementById('t1').firstChild.nodeName
4. document.getElementById('t1').firstChild.firstChild.nodeName
5. document.getElementById('t1').firstChild.firstChild.nodeValue

- Example 1 returns "ul"
- Example 2 returns "null"
- Example 3 returns "li"
- Example 4 returns "text"
  - A text node below the "li" which holds the actual text data as its value
- Example 5 returns " Item 1 "

Sample HTML

```html
<ul id="t1">
  <li> Item 1 </li>
</ul>
```
Page Manipulation with JavaScript

• Some possibilities
  – createElement(elementName)
  – createTextNode(text)
  – appendChild(newChild)
  – removeChild(node)

• Example: add a new list item

```javascript
var list = document.getElementById('t1')
var newitem = document.createElement('li')
var newtext = document.createTextNode(text)
list.appendChild(newitem)
newitem.appendChild(newtext)
```

Sample HTML
```html
<ul id="t1">
  <li> Item 1 </li>
</ul>
```
All the functional pieces are in place

• Now we can create personalized and dynamic websites. Yay!

• But what about security?
  – How do we stop websites from snooping around in each other’s business?
Goals of Web Security

• Safely browse the Web
  – A malicious website cannot steal information from or modify legitimate sites or otherwise harm the user...
    • ... even if visited concurrently with a legitimate site - in a separate browser window, tab, or even iframe on the same webpage
    • Based on Same Origin Policy (SOP)
  – A malicious website cannot steal or modify information on the local machine, nor can it interact in any way with local applications
    • Based on JavaScript safety and web browser design and implementation (Browser security)
Web Security Concerns

- Web Security is concerned with ensuring the following 3 properties for web applications:

  - **Authentication**: securely identify users on top of HTTP, which is a stateless protocol.
  - **Confidentiality**: protect any sensitive data that websites serve to the browser from other websites, and the user's own sensitive data outside the browser from any website.
  - **Integrity**: ensure that the data and the code served to users cannot be tampered with.
Authentication Methods

- HTTP authentication: username/passwd supplied in HTTP header
- Cookie authentication (most common):
  - username/password (login credentials) requested via a HTML form
  - server checks the credentials and then sets a cookie that identifies the user and his/her successful login
  - Browser returns this cookie with each subsequent request
- Hidden-form authentication
  - Similar to cookie authentication, but the server includes the session info in a hidden form field.
HTTP is a stateless protocol.

- User Authentication: Use cookies and send them implicitly for convenience.
- Server Authentication: SSL + Certification Authorities

Cookie-Based Authentication
Lifetime of Cached Cookies and HTTP Authentication Credentials

• Temporary cookies cached until browser shut down, persistent ones cached until expiry date

• HTTP authentication credentials cached in memory, shared by all browser windows of a single browser instance

• Caching depends only on browser instance lifetime, not on whether original window is open
Confidentiality

• No mutual trust among parties.
• Confidentiality through Isolation: Same-Origin Policy (SOP)
  ▪ Partition the Web into domains and isolate sensitive data such as cookie, network data and DOM nodes.
All of These Should Be Safe

- Safe to visit an evil website
- Safe to visit two pages at the same time
- Safe delegation
Same-Origin Policy (SOP)

- The SOP partitions the web into domains (according to their DNS origin) and isolates sensitive data from scripts running in other domains.
- What is sensitive data?
  - Cookies
  - Web page content (DOM isolation)
  - Web site response (Network isolation)
SOP: Cookie Isolation

- Each domain has its own set of independently managed cookies, and these are embedded only in requests to the same domain.
- Only scripts running from the same domain and responses from the same domain can read and write cookies.
- HTTP-Only cookies
SOP: Page content isolation

• Basic unit of isolation in a browser is a `<frame>`
  – `document.write` – refers to the current frame

• DOM Isolation
  ▪ Scripts only have access to DOM elements on the same domain.
  ▪ Frames embedded in a page are part of the DOM tree of the parent, but the policy still applies:
    ▪ `document.frames[0].title`
    ▪ Only accessible if the parent is from the same origin.
Domains vs Subdomains

• Subdomains
  – E.g. `private.example.com` vs `forum.example.com`
  – Considered different origin
  – Possibility to relax the origin to `example.com` using `document.domain`
  – Possibility to use cookies on `example.com`

• Completely separate domains
  – E.g. `private.example.com` vs `exampleforum.com`
  – Considered different origin, without possibility of relaxation
  – No possibility of shared cookies
SOP: Network isolation

• Script can send requests to arbitrary sites
• But scripts cannot read responses from any server
  ▪ They can still send blind requests to other domains.
  ▪ Is it safe for a malicious script to issue a request if it cannot read the response?
  ▪ CSRF (discussed later)

• Exception: XmlHttpRequests (XHR) permit a script to read from its origin server
Embedding and SOP: Caveats

- For embedded content, origin of the content may be different from the domain used for SOP checks
  - Scripts retrieved from B and embedded in A run with A privileges.
  - Akin to user A running an executable written by B in a UNIX environment.
    - Cross-site scripting attacks exploit this!
    - as do script inclusion attacks!
- Plugins implement their own SOP-like policies.
  - Flash keeps its server origin.
Same-Origin Policy: Exceptions

- Some resources are not considered sensitive and can be accessed across domains
  - Browser History: CSS allows website to use different rules for visited and unvisited links.
  - CSS rules: they can be read even when importing a cross-origin stylesheet
- Unsurprisingly, two attacks use these exceptions for information leaks
  - Cross-origin CSS and CSS history hacks exploit these exceptions
Limitations of SOP

- A very rigid policy that imposes an all-or-nothing approach:
  - The developer can embed the resource (allow all) or open it in an iframe (allow none).
  - Cannot import script libraries without trusting them blindly.
- Does not limit outgoing requests
Confidentiality (OS)

• Users do not trust the websites they visit.
• Again: Confidentiality through Isolation
• Sandboxing: only expose a safe API to web application that limits their interaction with the operating system
Integrity

• Network data integrity: HTTPS/DNSSEC
  • Also used to authenticate the server (e.g. Banks) and ensure network confidentiality.
  • Public-key protocol used to establish a session key to encrypt traffic.

• Browser data integrity: SOP
  • Think of integrity as write access on confidential resources. SOP protects from read as well as write accesses
Despite the same origin policy

• Many things can go wrong at the client-side of a web application

• Popular attacks
  – Cross-site Scripting
  – Cross-site Request Forgery
  – Session Hijacking
  – Session Fixation
  – SSL Stripping
  – Clickjacking
Where Does the Attacker Live?
Threat Model 1: Web Attacker

• User, network and server are benign

• Attacker controls a malicious website (attacker.com)
  – Can even obtain an SSL/TLS certificate for his site ($0)

• Entices user to visit attacker.com
  – Phishing email, enticing content, search results, placed by an ad network, blind luck ...
  – Attacker’s Facebook app

• Attacker has no other access to user machine!

• Variation: “iframe attacker”
  – An iframe with malicious content included in an otherwise honest webpage
    • Syndicated advertising, mashups, etc.
Attacks on Authentication

• CSRF and Clickjacking
  • Confused deputy attacks that cause the victim browser to send authenticated requests for the attacker's benefit
  • CSRF: Cross-site request forgery: attacker sends requests to another web site, impersonating browser user
  • Clickjacking: User intends to click on one link, but the browser recognizes a link on another site
    • Achieved using overlaid frames and by manipulating visibility related attributes
CSRF
Cross-site Request Forgery (CSRF)

<form method="POST" action="/changepass">
...
New Password: <input type="password" name="password">
</form>

» Browser makes the following request:

GET http://www.example.com/changepass?password=newpassword HTTP 1.1

» Let’s say the application didn’t authenticate password change request using any means other than cookies

» An attacker can easily forge request!

» Attack works because (a) cookies are sent by default, and (b) SOP does not restrict cross-origin submissions
POST Example

• POST requests can also be forged
• Attacker lures the client to visit his/her web page
  <iframe name="hiddenframe" style="display:none">
    <form method="POST" name="evilform" target="hiddenframe" action="http://www.examplewebsite.com/update_password">
      <input type="hidden" name="password" value="evilhax0r">
    </form>
    <script>document.evilform.submit()</script>
  </iframe>
CSRF and Authentication status

• The classic CSRF attack abuses a user’s existing session cookies with a victim website
• Does that mean that CSRF is a non-issue when a user is logged out?
  • No! (although many still think “yes”)
  • In certain cases, an attacker can log in a victim with his credentials using an unprotected login form and still manage some sort of abuse
    • Login CSRF
Possible targets of CSRF

- Banks
  - Attacker can issue a request to transfer money from victim’s bank account to attacker’s
- E-commerce sites
  - Purchase items using victim’s account, ship to attacker
- Forums and Social network sites
  - Post articles using victim’s identity
- Home/Intranet firewall
  - Reconfigure firewall to permit connections from the Internet to a host behind the firewall
- Note that victim user’s location is exploited: the attacker (typically) cannot communicate with the firewall, but the user’s browser can.
Preventing CSRF

• HTTP requests originating from user action are indistinguishable from those initiated by attacker

• Need methods to distinguish valid requests
  – Inspecting Referrer Headers
  – Validation via User-Provided Secret
  – Validation via Action Token
Inspecting Referrer Headers

• Referrer header specifies the URI of document originating the request

• Assuming requests from our site are good, don’t serve requests not from our site

• Unfortunately, Referrer information may be suppressed by browsers (or firewalls) for privacy reasons
Validation via User-Provided Secret

- Can require user to enter secret (e.g. login password) along with requests that make server-side state changes or transactions

- Example: The change password form could ask for the user’s current password

- Security vs convenience: use only for infrequent, “high-value” transactions
  - Password or profile changes
  - Expensive commercial/financial operations
Validation via Action Token

• Add special action tokens as hidden fields to authorized forms to distinguish from forgeries
• Need to generate and validate tokens so that malicious 3rd party can’t guess or forge token
  • Token should be a nonce that is unpredictable
  • Same-origin policy prevents 3rd party from inspecting the form to find the token
• This token can be used to distinguish genuine and forged forms
Clickjacking

Win a free iphone!
Just click on red and green!

Quick while the offer lasts!
So you click...

• Nothing happens.
  – Or something happens
  – But you don’t get that free iphone that you were promised

• Continue browsing

• Time to check email
  – Go to GMail
Where are my mails bro?!?
Win a free iphone!
Just click on red and green!

Quick while the offer lasts!
Win a free iphone!
Just click on red and green!
Quick while the offer lasts!
Clickjacking Defenses

• Disallow hidden frames
  • There are many ways to make a frame imperceptible

• Restrict framing
  • X-Frame-Options header
    • SAMEORIGIN;
    • Allow-from <uri>;
    • DENY;

• Content security policy (supercedes X-frame)
  – Content-Security-Policy: frame-ancestors ‘self’
  – Content-Security-Policy: frame-ancestors a.com b.org
  – Content-Security-Policy: frame-ancestors 'none'
Cross-Site Scripting (XSS)

• Different types of script injection
  – **Reflected**: part of the URI used in the response
  – **Persistent**: stored data used in the response
  – **DOM-based**: data used by client-side scripts
What can an attacker do with XSS?

• Long answer (non exhaustive):
  – Exfiltrate your cookies (session hijacking)
  – Make arbitrary changes to the page (phishing)
  – Steal all the data available in the web application
  – Make requests in your name
  – Redirect your browser to a malicious page
  – Tunnel requests to other sites, originating from your IP address (BEEF)
Reflected XSS Example

• Host www.vulnerable.site displays name submitted using a web form

• With benign data, following request may result

   GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0

• And the web site responds

   <HTML>
   <Title>Welcome!</Title>
   Hi Joe Hacker<br>
   Welcome to our system
   </HTML>

• What if the attacker submits

   GET welcome.cgi?name=<script>...<script> HTTP/1.0
Reflected XSS Summary

• Attacker causes victim to click on maliciously crafted link
  • Typically contains a malicious script as a parameter
• request goes to vulnerable web site
• web site does not properly check its input
• returns a page that contains the malicious script
  • which operates with privileges of the vulnerable site
    • can perform any action that the user can perform
      • send the cookie (or other private info) to the attacker
      • perform sensitive action, e.g., withdraw money
Persistent XSS

- Malicious script permanently stored on server
- Still requires
  - An attack that causes the script to be stored
  - Script should be used in a page visited by victim user
- User totally unaware of the vulnerability/exploit
  - More stealthy, damaging and long-lasting
  - How can this be possible?
    - Think of a blog, or social networking web site: input from one user is rendered in the page shown to another
DOM-Based XSS

• DOM-Based refers to how the script comes about
  • Plain XSS: malicious script is already present in the page from server
  • DOM-based XSS:
    • server delivers an initial page content and a legitimate script
    • execution of this script constructs the rest of the page using DOM operations
      • document.write
      • document.createElement
      • document.appendChild ...
    • malicious script content manifests during this construction
• Orthogonal to reflected vs persistent categorization
  • DOM-based XSS can be of either kind
Preventing XSS

- Server should not send untrusted data to the browser that could result in the creation of an unintended (and unauthorized) script
  - Usually can just suppress certain characters, but this is not enough in the case of DOM-based

- We show examples of various contexts in HTML document as template snippets
  - Variable substitution placeholders: \%(var\)
  - evil-script; will denote what attacker injects
  - Contexts where XSS attack is possible
Most straightforward, common situation

Example Context:

Error: Your query '%(query)' did not return any results.

- Attacker sets query = <script>evil-script;</script>
- HTML snippet renders as

Error: Your query '<script>evil-script;</script>' did not return any results.

Prevention: HTML-escape untrusted data

Rationale: If not escaped
- <script> tags evaluated, data may not display as intended
Tag Attributes (e.g., Form field values)

- Contexts where data is inserted into tag attribute
- Attacker able to “close the quote”, insert script
- Example HTML Fragment:

  ```html
  <form ...><input name="city" value="%{city}"/>
  <script>evil-script;</script>
  </form>
  ```

  - Attacker sets
    ```html
    city = xyz"<script>evil-script;</script>
    ```

  - Renders as
    ```html
    <form ...
    <input name="city" value="xyz">
    <script>evil-script;</script>
    </form>
    ```
More Attribute Injection Attacks

- **Image Tag**: `<img src=%(image_url)>`
- **Attacker sets** `image_url = http://www.example site.org/onerror=evil-script;`
- **After Substitution**: `<img src=http://www.example site.org/onerror=evil-script;>`
  - Lenient browser: first whitespace ends `src` attribute
  - `onerror` attribute sets handler to be desired script
  - Attacker forces error by supplying URL w/o an image
  - Can similarly use `onload`, `onmouseover` to run scripts
  - Attack string didn’t use any HTML metacharacters!
Dynamic URL attributes vulnerable to injection

Script/Style Sheet URLs: `<img src="% (script_url)">

- Attacker sets `script_url = http://hackerhome.org/evil.js`

javascript: URLS - `<img src="%(img_url)">

- By setting `img_url = javascript:evil-script;` we get `<img src="javascript:evil-script;">`

- And browser executes script when loading image
**Style Attributes**

- **Dangerous if attacker controls style attributes**
  - Attacker injects:
  - Browser evaluates:
    ```html
    <div style="background: %(color)">I like colors.</div>
    ```
    ```html
    color = green; background-image: url(javascript:evil-script;)
    ```
    ```html
    <div style="background: green; background-image: url(javascript:evil-script;)
    I like colors. </div>
    ```

- **In IE 6 (but not Firefox 1.5), script is executed!**

- **Prevention: whitelist through regular expressions**
  - Ex: `^([a-z]+)|(#\d{0,6})+$` specifies safe superset of possible color names or hex designation
In JavaScript Context

- Be careful embedding dynamic content
- `<script>` tags or handlers (onclick, onload,...)

Attacker injects:

```javascript
var msg_text = 'oops';
evil-script; //'
// do something with msg_text
```

And evil-script; is executed!
Another JavaScript Injection Example

- From previous example, if attacker sets
  ```html
  msg_text = foo</script><script>evil-script;</script><script>
  ```
  the following HTML is evaluated:
  ```html
  <script>var msg_text = 'foo</script>
  <script>evil-script;</script>
  <script>'// do something with msg_text</script>
  ```

- Browser parses document as HTML first
- Divides into 3 `<script>` tokens before interpreting as JavaScript
- Thus 1\textsuperscript{st} & 3\textsuperscript{rd} invalid, 2\textsuperscript{nd} executes as evil-script
Defending against XSS

• Blacklisting
  – E.g. No <, >, script, document.cookie, etc.
  – Intuitively correct, but it should **NOT** be relied upon
    • As we saw in the last few slides, there are too many ways to insert script content. The XSS Cheat Sheet lists hundreds of possibilities

• Whitelist whenever possible
  – E.g. this field should be a number, nothing more nothing less

• Always escape user-input
  – Neutralize “control” characters for all contexts

• Content Security Policy
  – Whitelist for resources
  – Belongs in the “if-all-else-fails” category of defense mechanisms
Content Security Policy

• Example

```
Content-Security-Policy: default-src https://cdn.example.net; frame-src 'none'; object-src 'none'; image-src self;
```

• CSP is very powerful
  – Great if you are writing something from scratch
  – Not so great if you have to rewrite something to CSP
    • E.g. Convert all inline JavaScript code to files
Content Security Policy v2

• CSP was great in theory but still hasn’t caught up in practice

• CSP v2.0 supports two new features to help adopt CSP
  – Script nonces for inline scripts
  – Hashes for inline scripts
  – Read more here:
    • https://blog.mozilla.org/security/2014/10/04/csp-for-the-web-we-have/
Content Security Policy v2

Script nonces for inline scripts

[HTTP Header] Content-security-policy: default-src 'self';
   script-src 'nonce-2726c7f26c'

[HTML] <script nonce="2726c7f26c">... </script>

Hashes for inline scripts

[HTTP Header] content-security-policy: script-src 'sha256-cluU6nVzrYJlo7rUa6TMmz3nyIPFrPQrEUpOHllb5ic='

[HTML] <script> ... </script>
Browser XSS filters

• Some browsers try to help by attempting to detect *reflected XSS* and stop them
  – Internet Explorer was the first to introduce this
  – Chrome followed a bit later, with a more complete approach that addressed some of IEs problems
    • Unfortunately, over the years, Chrome’s filter seems to have gone back on some of the improvements. Its filter stops fewer attacks than IE in our experiments
  – Firefox invested in an XSS filter for some time, but then seems to have abandoned its efforts
    • PaleMoon, a Firefox clone, imported the XSS filter for Firefox developed at Stony Brook.
Browser XSS filters

Attempt 1: Use string (or regexp) matching to identify suspicious content within request parameters (NoScript)
Example: excise “<script>”, “data:”, etc. from parameters
Problem: High False Positives make it unsuitable for general use

Attempt 2: Filter only if suspicious parameter is reflected, i.e., its value appears in the HTML response (IE/Edge)
FPs can still be too high
Mitigate using very strict matching rules (IE/Edge, Chrome)
Unfortunately, this leads to false negatives and filter evasion

Attempt 3: Filter if suspicious reflected content is used in a dangerous context in the HTML response (Firefox filter)
Example: “data:” can safely appear outside HTML tags
In our filter, this reduces FPs sufficiently to enable use of approximate matching
Result: Evasion resistant XSS filtering
Script Inclusion

• What if an attacker can’t find an XSS vulnerability in a website
  – Can he somehow still get to run malicious JavaScript code?

• Perhaps... by abusing existing trust relationships between the target site and other sites
JavaScript libraries

• Today, a lot of functionality exists, and all developers need to do is link it in their web application
  – Social widgets
  – Analytics
  – JavaScript programming libraries
  – Advertising
  – ...

Remote JavaScript libraries

<html>
...  
<script src="http://www.foo.com/a.js"> </script>
...  
</html>

• The code coming from foo.com will be incorporated in mybank.com, as if the code was developed and present on the servers of mybank.com
Remote JavaScript libraries

• This means that if, foo.com, decides to send you malicious JavaScript, the code can do anything in the mybank.com domain

• Why would foo.com send malicious code?
  – Why not?
  – Change of control of the domain
  – Compromised
Timing attacks

• Because of the same-origin policy, scripts cannot access most resources in a cross-domain
  – Can still make the requests though, that’s why CSRF is a problem

• An attacker can still abuse the time it takes for a page to load, as a side-channel
Timing attacks

• Scenario: I want to know if you are logged into your Gmail
  – I may, or may not be able to load the page in an iframe, depending on the Xframe-options
  – Even if I can load it, I still can’t peek in it

• What if I try to load mail.google.com as an image?
  – `<img src="https://mail.google.com" onError="func()"/>
  – The browser will fetch the page with your cookies and then the parser will at some point throw an error that this is not an image
Timing attacks

• The size of a page is often dependent on whether you are logged in or not

• Hence, for a large image, the browser will take a longer time to give you an error

• Oversimplified attack:
  – Fast error: not-logged in
  – Slow error: logged-in
Getting one measurement

```html
<html><body><img id="test" style="display: none">
<script>
    var test = document.getElementById('test');
    var start = new Date();
    test.onerror = function() {
        var end = new Date();
        alert("Total time: " + (end - start));
    }
    test.src = "http://www.example.com/page.html";
</script>
</body></html>
```

**Figure 3: Example JavaScript timing code**

Code sample from: *Exposing Private Information by Timing Web Applications*  
By Bortz et al.
Threat Model 2: Network Attacker
SSL Stripping

• Let’s say that a website exists only over HTTPS
  – No HTTP pages

• Two scenarios
  1. User types https://www.securesite.com and the browser directly tries to communicate the remote server over a secure channel
  2. User types http://www.securesite.com (or just securesite.com) and the site will redirect the user to the secure version (using an HTTP redirection/Meta header)
Normal page load
Page load when attacker is present
SSL Stripping

• Same thing can happen when sites deliver HTTPS-targeted forms over an HTTP connection (typically for performance or outsourcing purposes)

```html
<form action=https://example.com/login>
  <input .... username>
  <input .... password>
</form>
```
Defenses

- Use full-site SSL in combination with Secure cookie and HTTP-only Cookie

- HSTS: HTTP Strict Transport Security
  - Force the browser to always contact the server over an encrypted channel, regardless of what the user asks

HTTP Header

```
Strict-Transport-Security: max-age=31536000
```
Defenses

• What about the very first time you visit a website?
  – What if a MITM is located on your network and will therefore strip SSL and suppress HSTS?

• Answer:
  – Preloaded HSTS: Websites can ask browsers to mark them as HSTS in a special browser-vendor-updated database
Threat model 3: Malicious Client

• In these scenarios:
  – The server is benign
  – The client is malicious
    • The client can send arbitrary requests to the server, not bound by the HTML interfaces

• The attacker is after information at the server-side
  – Steal databases
  – Gain access to server
  – Manipulate server-side programs for gain
### OWASP Top 10

<table>
<thead>
<tr>
<th>A1 - Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 - Broken Auth and Session Management</td>
</tr>
<tr>
<td>A3 - Cross-site Scripting</td>
</tr>
<tr>
<td>A4 - Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 - Security misconfiguration</td>
</tr>
<tr>
<td>A6 - Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7 - Missing function level access control</td>
</tr>
<tr>
<td>A8 - Cross-site Request Forgery</td>
</tr>
<tr>
<td>A9 - Using components with kn. vulnerabilities</td>
</tr>
<tr>
<td>A10 - Unvalidated redirects and Forwards</td>
</tr>
</tbody>
</table>
Injection Attacks

• SQL injection
  • Steal sensitive data about specific user
  • All username, password (hashes) info
  • ...

• Command injection
  • Install malware on server, run reconnaissance commands, probe serverside network, inject into command streams for backend servers, ...

• We discussed these attacks and their defenses before
  • Defenses need to be mindful of trust boundaries, e.g., don’t rely on client-side sanitization if the attacker is the client!
Redirects, Cookies, and Header Injection

- Need to filter and validate user input inserted into HTTP response headers
- Ex: servlet returns HTTP redirect

HTTP/1.1 302 Moved
Content-Type: text/html; charset=ISO-8859-1
Location: %(redir_url)s

<html>
<head><title>Moved</title></head>
<body>Moved <a href='%(redir_url)s'>here</a></body>
</html>

- Attacker Injects: (URI-encodes newlines)
  oops:foo\r\n  \n  Set-Cookie: SESSION=13af..3b;
  domain=mywwwservice.com\r\n  <script>evil()</script>
Logic Vulnerabilities

• HTTP parameter tampering vulnerabilities are a subset of logic vulnerabilities in web applications

• Logic vulnerabilities typically rely on breaking assumptions made by architects and developers

• Assumptions
  – Step 2 can only be performed after Step 1
  – The web application controls the navigation steps
  – Users cannot change parameters that they cannot see
  – Etc.
Examples of logic vulnerabilities

• Unlike vulnerabilities discussed so far, logic vulnerabilities don’t have a clear, narrow definition

• This makes them hard to identify, especially by automated vulnerability discovery tools

• We will see a few real-world examples based on the book “The Web Application Hackers Handbook”
Case Study: Password change

• A website allows its users to change their password, by filling out a form with their current password, and their new password

• Administrators can also change a user’s password but they don’t need to provide a user’s current password

```java
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
```
Case Study: Password change

• The code that handles these two cases is the same and the developer assumes that if the “existingPassword” parameter is not present, this must be because the current request came from an administrative UI

• All the attackers has to do is drop the “existingPassword” HTTP parameter from the outgoing request
Case Study: Bulk Discounts

• An online shop gives users discounts when they buy some products together
  – E.g. If you purchase an antivirus solution, and a personal firewall, and antispam software then you are entitled to 25% discount on each product

• Abuse
  – Add all products in your basket to get the discount and then remove the ones you don’t want
Case Study: Escaping from escaping

• A web application has to pass user-controllable input as an argument to an operating system command.
• The developer creates a list of special shell metacharacters that need escaping
  – ; | & < > ‘ space and newline
• If any of these are present in the input, the code escapes them by prepending them with a backslash
  – \
Case Study: Escaping from escaping

• If an attacker types
  – foo;ls
• The code converts it to
  – foo\;ls

• What if an attacker types an escape character
  – foo\;ls
• Will become
  – foo\\;ls
• Which amounts to escaping the backslash but not the semicolon
Weaknesses Leading to Attacks

- Trusting embedded content
  - Embedded scripts have same privilege as surrounding page (XSS)
  - Embedded content can target browser flaws, e.g., buffer overflows

- Not restricting outgoing network requests
  - Unauthorized requests to third-party sites (CSRF)
  - Include trusted party content in a frame
    - Abuse trust in third party, e.g., to improve odds of successful phishing
    - Clickjacking
  - Attacking third-party sites, e.g., portscanning or launching exploits
  - Ease of leaking sensitive data acquired (e.g., send cookie to attacker)

- Allowing Turing-complete computation for arbitrary sites
  - Bitcoin mining
  - Side-channel attacks
  - JIT-ROP attacks

- Weaknesses in lower layers
  - In-network attacks, e.g., man-in-the-middle
  - DNS compromise

- Application development environments that blur trust boundaries
  - Trusting client-side: browser and/or scripts running on a web page (Parameter tampering, ...)

- Good old application logic or implementation vulnerabilities
  - SQL injection, command injection, HTTP parameter pollution, ...
Credits

• Many of the slides here are the courtesy of Nick Nikiforakis and Venkat Venkatakrishnan
Memory Error Exploits and Defenses
### Process Memory Layout

<table>
<thead>
<tr>
<th>High Mem</th>
<th>Low Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv, env</td>
<td>text</td>
</tr>
<tr>
<td>stack</td>
<td>data</td>
</tr>
<tr>
<td>heap</td>
<td>bss</td>
</tr>
</tbody>
</table>

**Argv/Env:** CLI args and environment  
**Stack:** generally grows downwards  
**Heap:** generally grows upwards  
**BSS:** uninitialized global data  
**Data:** initialized global data  
**Text:** read-only program code
Memory Layout Example

/* data segment: initialized global data */
int a[] = { 1, 2, 3, 4, 5 };
/* bss segment: uninitialized global data */
int b;

/* text segment: contains program code */
int main(int argc, char **argv) /* ptr to argv */
{
    /* stack: local variables */
    int *c;
    /* heap: dynamic allocation by new or malloc */
    c = (int *)malloc(5 * sizeof(int));
}
What is the Call Stack?

LIFO data structure: push/pop
- Stack grows downwards in memory.
- SP (esp) points to top of stack (lowest address)

What’s on the call stack?
- Function parameters
- Local variables
- Return values
- Return address
Call Stack Layout

```plaintext
b() {
  ...
}

a() {
  b();
}

main() {
  a();
}
```
Accessing the Stack

Pushing an item onto the stack.
1. Decrement SP by 4.
2. Copy 4 bytes of data to stack.
Example: push 0x12

Popping data from the stack.
3. Copy 4 bytes of data from stack.
4. Increment SP by 4.
Example: pop eax
Retrieve data without pop: mov eax, esp
What is a Stack Frame?

Block of stack data for one procedure call.

Frame pointer (FP) points to frame:

- Use offsets to find local variables.
- SP continually moves with push/pops.
- FP only moves on function call/return.
- Intel CPUs use ebp register for FP.
C Calling Convention

1. Push all params onto stack in reverse order.
   Parameter #N
   ...
   Parameter #2
   Parameter #1

2. Issues a call instruction.
   1. Pushes address of next instruction (the return address) onto stack.
   2. Modifies IP (eip) to point to start of function.
Stack before Function Executes

<table>
<thead>
<tr>
<th>old stack frame</th>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter #N</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>parameter #1</td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td>Stack Pointer</td>
</tr>
</tbody>
</table>
C Calling Convention

1. Function pushes FP (ebp) onto stack.
   Save FP for previous function.
   push ebp

2. Copies SP to FP.
   Allows function to access params as fixed indexes from base pointer.
   mov ebp, esp

3. Reserves stack space for local vars.
   subl esp, 0x12
<table>
<thead>
<tr>
<th>Stack at Function Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>old stack frame</td>
</tr>
<tr>
<td>parameter #N</td>
</tr>
<tr>
<td>…</td>
</tr>
<tr>
<td>parameter #1</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>old FP</td>
</tr>
<tr>
<td>Space for local vars</td>
</tr>
<tr>
<td>Space for local vars</td>
</tr>
</tbody>
</table>

EBP (Base Pointer)  
ESP (Stack Pointer)
C Calling Convention

1. After execution, stores return value in eax.
   movl eax, 0x1
   Resets stack to pre-call state.
   Destroys current stack frame; restores caller’s frame.
   mov esp, ebp
   pop ebp

2. Returns control back to where called from.
   ret pops top word from stack and sets eip to that value.
Example: Stack Smashing Attack

```c
void f(const int *A, int n) {
    int buf[100];
    int i = 0;
    while (i < n) {
        buf[i] = A[i++];
    }
    ...
}
```

Injected code starts here
Stack smashing defenses

- Canary stored before return value, checked before return
  - Issues
    - Protecting RA vs Saved BP
    - Random, XOR, null canaries
    - How about data?
  - Weaknesses
    - Brute-force canary, or rely on information leakage attacks
    - Overwrite RA without overwriting canary (e.g., double pointer attacks)
    - Overwrite other code pointers (e.g., function pointer, virtual table pointer, GOT)

- Storing RA in two places
  - StackShield, Return address defender (RAD)
  - Issues: compatibility with signals, exceptions, longjmp

- Propolice
  - Canary before saved BP + protect local variables by reordering them
    - Simple variables (integers, pointers) located at lower addresses, buffers at higher addresses
      - Buffer overflow cannot corrupt local variables, preventing double pointer attacks
      - But underruns can corrupt these simple (non-buffer) variables
  - Mainstream compilers (gcc, MS) include Propolice like protection
    - Not included for functions with no arrays
Non-executable data

Direct code injection attacks at some point execute data

- Most programs never need to do this

Hence, a simple countermeasure is to mark data memory (stack, heap, ...) as non-executable

- Write-XOR-Execute, DEP

This counters direct code injection

- In principle, this countermeasure may also break certain legacy applications
Instead of injecting malicious code, why not assemble malicious code out of existing code already present in the program

- *Indirect code injection* attacks will drive the execution of the program by manipulating the stack

**E.g.** Just execute `system("/bin/bash")` instead of creating your own interrupts

- You just need to find where the system function is and call it with the right parameter
Return-into-libc: overview

Code Memory

<table>
<thead>
<tr>
<th>f1</th>
<th>.</th>
<th>.</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td>.</td>
<td>.</td>
<td>return</td>
</tr>
<tr>
<td>f3</td>
<td>return</td>
<td>.</td>
<td>return</td>
</tr>
</tbody>
</table>

Stack

| Params for f1 |
| Return addr  |
| Params for f2 |
| Return addr  |
| Params for f3 |
| Return addr  |

SP → IP
Return-into-libc: overview
Return-into-libc: overview
Return-into-libc: overview
Return-into-libc: overview

Stack

- Params for f1
- Return addr
- Params for f2
- Return addr

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return

SP

IP
Return-into-libc: overview

Stack

Params for f1

Return addr

Params for f2

Return addr

Code Memory

f1

return

f2

return

f3

return

IP

SP
Return-into-libc: overview

Stack

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return

SP

Return addr

Params for f1

IP
Return-to-libc

What do we need to make this work?

• Inject the fake stack
  • Easy: this is just data we can put in a buffer

• Make the stack pointer point to the fake stack right before a return instruction is executed

• Then we make the stack execute existing functions to do a direct code injection
  • But we could do other useful stuff without direct code injection
Overwritten saved EIP need not point to the beginning of a library routine

Any existing instruction in the code image is fine
  • Will execute the sequence starting from this instruction

What if instruction sequence contains RET?
  • Execution will be transferred... to where?
  • Read the word pointed to by stack pointer (ESP)
    • Guess what? Its value is under attacker’s control! (why?)
  • Use it as the new value for EIP
    • Now control is transferred to an address of attacker’s choice!
  • Increment ESP to point to the next word on the stack
Can chain together sequences ending in RET
  • Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

What is this good for?

Answer [Shacham et al.]: everything
  • Turing-complete language
  • Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  • Attack can perform arbitrary computation using no injected code at all – return-oriented programming
## Return Oriented Programming

<table>
<thead>
<tr>
<th>High</th>
</tr>
</thead>
</table>
| ...
| ...
| ...
| 0x80abdea0 |
| 0x309 |
| 0x80345677 |
| "/tmp/lala" |
| 0x80abddaa |
| 8 |
| 0x80abcdee |

<table>
<thead>
<tr>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
</tr>
</tbody>
</table>

| EAX = SMTH |
| EBX = SMTH |
| ECX = SMTH |

```assembly
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...```
Return Oriented Programming

EAX = SMTH
EBX = SMTH
ECX = SMTH

ESP

EIP

High

0x80abdea0
0x309
0x80345677
"/tmp/lala"
0x80abddaa
8
0x80abcdee

Low

EIP

0x80345677: pop $ecx;
0x80345678: ret;

0x08abcdee: pop $eax;
0x08abcdef: ret;

0x80abddaa: pop $ebx;
0x80abddab: ret;

0x80abdea0: int 0x80;
Return Oriented Programming

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>0x80abdea0</td>
</tr>
<tr>
<td>...</td>
<td>0x309</td>
</tr>
<tr>
<td>...</td>
<td>0x80345677</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>&amp;”/tmp/lala”</td>
</tr>
<tr>
<td>8</td>
<td>0x80abcdee</td>
</tr>
</tbody>
</table>

EAX = 8  
EBX = SMTH  
ECX = SMTH

0x80abdea0: int 0x80;
0x80abdeaa: pop $ebx;  
0x80abddab: ret;

0x80abcdee: pop $eax;  
0x80abcdef : ret;

0x80abddaa: pop $ebx;  
0x80abddab: ret;

0x80345677: pop $ecx;  
0x80345678: ret;
Return Oriented Programming

EAX = 8
EBX = SMTH
ECX = SMTH

0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

0x80345677: pop $ecx;
0x80345678: ret;

0x08abcdee: pop $eax;
0x08abcdef : ret;

0x80abddaa: pop $ebx;
0x80abddab: ret;

0x80abdea0: int 0x80;
...
**Return Oriented Programming**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td>0x80abdea0</td>
</tr>
<tr>
<td>0x309</td>
<td>0x309</td>
</tr>
<tr>
<td>0x80345677</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>0x80abddaa</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0x80abcdee</td>
<td>0x80abcdee</td>
</tr>
</tbody>
</table>

**EAX = 8**
**EBX = &"/tmp..."**
**ECX = SMTH**

```
0x80345677: pop $ecx;
0x80345678: ret;

0x08abcdee: pop $eax;
0x08abcdef : ret;

0x80abddaa: pop $ebx;
0x80abddab: ret;

0x80abdea0: int 0x80;
```
Return Oriented Programming

<table>
<thead>
<tr>
<th>ESP</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td>...</td>
</tr>
<tr>
<td>0x309</td>
<td>...</td>
</tr>
<tr>
<td>0x80345677</td>
<td>...</td>
</tr>
<tr>
<td>&amp;&quot;/tmp/lala&quot;</td>
<td>...</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td>0x80abcdee</td>
<td>...</td>
</tr>
<tr>
<td>0x80abdeea0</td>
<td>...</td>
</tr>
<tr>
<td>0x80345678: ret;</td>
<td>...</td>
</tr>
</tbody>
</table>

EAX = 8
EBX = &"/tmp...
ECX = SMTH

...
Return Oriented Programming

<table>
<thead>
<tr>
<th>ESP</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td>0x80345677</td>
</tr>
<tr>
<td>0x309</td>
<td>&amp;”/tmp/lala”</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>8</td>
</tr>
<tr>
<td>0x80abcdee</td>
<td>0x80abdea0</td>
</tr>
</tbody>
</table>

EAX = 8  
EBX = &”/tmp...”  
ECX = 0x309  

0x80345677: pop $ecx;  
0x80345678: ret;  

0x08abcdef: pop $eax;  
0x08abdddf: ret;  

0x80abddaa: pop $ebx;  
0x80abddab: ret;  

0x80abdea0: int 0x80;
Return Oriented Programming

ESP

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x80abdea0</td>
<td>0x309</td>
</tr>
<tr>
<td>0x80345677</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>8</td>
</tr>
<tr>
<td>0x80abcdee</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EAX = 8
EBX = &"/tmp..."
ECX = 0x309

... 0x80345677: pop $ecx; 0x80345678: ret;
... 0x08abcdee: pop $eax; 0x08abcdef : ret;
... 0x80abddaa: pop $ebx; 0x80abddab: ret;
... 0x80abdea0: int 0x80; ...
**Heap based buffer overflow**

If a program contains a buffer overflow vulnerability for a buffer allocated on the heap, there is no return address nearby.

So attacking a heap based vulnerability requires the attacker to overwrite other code pointers.

We look at two examples:

- Overwriting a function pointer
- Overwriting heap metadata
Overwriting a function pointer

Example vulnerable program:

typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
Overwriting a function pointer

And what happens on overflow:

<table>
<thead>
<tr>
<th>buff (char array at start of the struct)</th>
<th>(cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: 0x00353068 0x0035306c 0x00353070 0x00353074</td>
<td>0x00353078</td>
</tr>
<tr>
<td>content: 0x656c6966 0x662f2f3a 0x61626f6f 0x00000072</td>
<td>0x004013ce</td>
</tr>
</tbody>
</table>

(a) A structure holding “file://foobar” and a pointer to the `strcmp` function.

<table>
<thead>
<tr>
<th>buff (char array at start of the struct)</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: 0x00353068 0x0035306c 0x00353070 0x00353074</td>
<td>0x00353078</td>
</tr>
<tr>
<td>content: 0x656c6966 0x612f2f3a 0x61666473 0x61666473</td>
<td>0x00666473</td>
</tr>
</tbody>
</table>

(b) After a buffer overflow caused by the inputs “file://” and “asdfsdfasdfs”.

<table>
<thead>
<tr>
<th>buff (char array at start of the struct)</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: 0x00353068 0x0035306c 0x00353070 0x00353074</td>
<td>0x00353078</td>
</tr>
<tr>
<td>content: 0xfeeb2ecd 0x11111111 0x11111111 0x11111111</td>
<td>0x00353068</td>
</tr>
</tbody>
</table>

(c) After a malicious buffer overflow caused by attacker-chosen inputs.
Overwriting heap metadata

The heap is a memory area where dynamically allocated data is stored

• Typically managed by a memory allocation library that offers functionality to allocate and free chunks of memory (in C: `malloc()` and `free()` calls)

Most memory allocation libraries store management information in-band

• As a consequence, buffer overruns on the heap can overwrite this management information

• This enables an “indirect pointer overwrite”-like attack allowing attackers to overwrite arbitrary memory locations
Dlmalloc maintains a doubly linked list of free chunks.

When chunk c gets unlinked, c’s backward pointer is written to *(forward pointer + 12)*.

Or: green value is written 12 bytes above where red value points.
Exploiting a buffer overrun

Top Heap grows with brk()

Green value is written 12 bytes above where red value points

A buffer overrun in d can overwrite the red and green values

- Make Green point to injected code
- Make Red point 12 bytes below a function return address
Exploiting a buffer overrun

Top Heap grows with brk()

Green value is written 12 bytes above where red value points

Net result is that the return address points to the injected code
Heap Overflows

- More generally, provides a primitive to write an arbitrary 32-bit value at an arbitrary location

- Possible targets
  - Function pointers
    - Return address on stack
      - Canaries don’t help, but second RA copy will detect attack
    - Global Offset Table (GOT)
  - Function pointers in static memory
  - Data pointers
    - Names of programs executed or files opened
    - Application-specific data, e.g., “is_authenticated” flag in a login-like program
Heap Overflow Defenses

- **Heap canaries**
  - “magic numbers” between data and header

- **Separation of metadata from data**
  - In general, separating control data from program data is a good idea
    - Helps prevent data corruption attacks from altering the control-flow of programs
  - Can be applied on the stack as well
    - “Safe stack” holds control-data
      - “safe” data (e.g., local integer-valued variables) can also be located there as they cannot be involved in memory errors
    - All other data moved to a second stack
Format-string Attacks

- **Exploits code of the form**
  - Read variables from untrusted source
  - `printf(s)`

- **Printf usually reads memory, so how can it be used for memory corruption?**
  - “%n” primitive allows for a memory write
  - Writes the number of characters printed so far (character count)
  - Many implementations (Linux, Windows) allow just the least significant byte of the number of character count
    - you don’t have to print large number of characters to write arbitrary 32-bit values --- just perform 4 separate writes of the LS byte of character count
    - Use field-width specifications to control character count

- **Formatguard: pass in actual number of parameters so the callee can only dereference that many parameters**
  - Not adopted in practice due to compatibility issues
**Integer Overflows**

- **There are multiple forms**
  - Assignment between variables of different width
    - Assign 32-bit value to 16-bit variable
  - Assignment between variables of different signs
    - Assign an unsigned variable to a signed variable or vice-versa
  - Arithmetic overflows
    - \( i = j+k \)
    - \( i = 4\times j \)
    - Note that \( i \) may become smaller than \( j \) even if \( j > 0 \)

- **Exploitation**
  - Allocate less memory than needed, leading to a heap overflow
    - One of the common forms of file-format attacks
  - “Escape” bounds checks
    - If \((i < \text{sizeof}(buf))\) `memcpy(buf, src, i)`;

- **For more info:**
  - [http://www.phrack.org/archives/60/p60-0x0a.txt](http://www.phrack.org/archives/60/p60-0x0a.txt)
Memory Errors

- Although other attack types have emerged, memory errors continue to be the dominant threat
  - Behind most “critical updates” from Microsoft and other vendors
  - Mechanism of choice in “mass-market” attacks, including worms
  - Evolved to target client (web browsers, email-handlers, word-processors, document/image viewers, media players, …) rather than server applications (e.g., web browsers)

- A memory error occurs when an object accessed using a pointer expression is different from the one intended
  - Spatial error
    - Examples
      - Out-of-bounds access due to pointer arithmetic errors
      - Access using a corrupted pointer
      - Uninitialized pointer access
  - Temporal error: access to objects that have been freed (and possibly reallocated)
    - Example: dangling pointer errors
    - applicable to stack and heap allocated data
Memory Errors in C

- **Spatial errors:** out-of-bounds subscript or pointer
  - `char *p = malloc(10); *(p+15);`

- **Temporal errors:** pointer target no longer valid
  - Uninitialized pointer
  - Dangling pointer
  - `free(p); q = malloc(...); *p;`
  - **Note:** target may be reallocated!

- **Hard to debug, especially temporal errors**
  - Unpredictable delay, unpredictable effect
  - **Reallocated pointer errors are the worst kind**
  - “Defensive programming” leads to memory leaks
Use of Memory Errors in Attacks

- Temporal errors
  - Not as frequently targeted as spatial errors, but are becoming more common (“double free,” “use-after-free”)

- Spatial errors
  - Pointer corruption is most popular
  - Out-of-bounds errors are most commonly used to corrupt pointers
    - But some attacks rely on just reads without necessarily corrupting existing data, e.g., heartbleed SSL vulnerability

- Typically, multiple memory errors (2 to 3) are used in an attack
  - Stack-smashing relies on out-of-bounds write, plus the use of a corrupted pointer as return address
  - Heap overflow relies on out-of-bounds write, use of corrupted pointer as target of write, and then the use of a corrupted pointer as branch target.
Overwrites aren’t the only problem...
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE?
IF SO, REPLY "POTATO" (6 LETTERS).

User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435.

User Ada sends this message: "H"
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e3cbb9ff89b13bff8)

HMM...

User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e3cbb9ff89b13bff8)

BIRD
SERVER, ARE YOU STILL THERE?
IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoRasT".

HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoRasT".
High-level Overview of Memory Error Defenses

❖ Block memory errors
  ▪ Bounds-checking (mainly focused on spatial error)
      ▼ Bounds-checking C and CRED, Valgrind memcheck, ...
      ▼ Blocking all memory errors (including temporal)

❖ Disrupt exploits
  ▪ Identify mechanisms used for exploit, block them
      ▼ Disrupt mechanism used for corruption
          – Protect attractive targets against common ways to corrupt them
            (“guarding” solutions)
      ▼ Disrupt mechanism used for take-over
          – Disrupt ways in which the victim program uses corrupted data
            – Randomization-based defenses
      ▼ Disrupt payload delivery mechanism
          – DEP, CFI
A. Disrupting Memory Error Exploits
1. Disrupting mechanisms used for corruption

- Stackguard and related solutions
  - Protect RA and saved BP; with ProPolice, some local variables as well
- Magic cookies and safe linking on heaps
- Attacks on GOT
  - GOT contains function pointers used to call library functions
    - Compiler generates a stub for each library function in a code section called PLT (program linkage table)
    - Stub code for a function f performs an indirect jump using the address stored in the GOT corresponding to f.
  - Defense: hide GOT
    - Not very effective: injected code can search and locate it!
- Common problem for this approach: incomplete
  - Not all targets can be protected
  - Incomplete even for protected targets: some corruption techniques can still succeed, e.g., corrupting RA without disturbing canary.
2. Disrupting payload delivery mechanisms

- Prevent control transfer to execution of injected code
  - Most OSes enforce \( W \oplus X \) (aka NX or DEP)
    - Prevents writable memory from being executable, so can’t execute injected code
  - Attackers get around this by reusing existing code
    - return-to-libc: return to the beginning of existing functions
      - Instead of having injected code spawning a shell, simply “return” to the execle function in libc
      - If it is a stack-smash, attacker controls the contents of the stack at this point, so they can control the arguments to execle
    - By constructing multiple frames on the stack, it is possible to chain together multiple fragments of existing code
      - ROP (return-oriented programming) takes this to the extreme
        - Chains together many small fragments of existing code (“gadgets”)
        - Each gadget can be thought of as an “instruction” for a “virtual machine”
        - For sufficiently complex binaries, sufficient number and variety of gadgets are available to support Turing-complete computation
      - Most exploits today rely on ROP, due to widespread deployment of \( W \oplus X \)
        - Goal of ROP payload is to invoke mprotect system call to disable \( W \oplus X \).
  - Control-flow integrity (CFI) is another (partial) defense that limits attacker’s freedom in terms of control transfer target
    - Can defeat most injected code and ROP attacks, but is not fool-proof
      - Skilled attackers may be able to craft attacks that operate despite CFI
3. Disrupting take-over mechanism

- **Key issue for an attacker:**
  - using attacker-controlled inputs, induce errors with predictable effects

- **Approach:** exploit software bugs to overwrite critical data, and the behavior of existing code that uses this data
  - Relative address attacks (RA)
    - Example: copying data from input into a program buffer without proper range checks
  - Absolute address attacks (AA)
    - Example: store input into an array element whose location is calculated from input.
      - Even if the program performs an upper bound check, this may not have the intended effect due to integer overflows
  - RA+AA attacks: use RA attack to corrupt a pointer $p$, wait for program to perform an operation using $*p$
    - Stack-smashing, heap overflows, ...
Disrupting take-over: Diversity Based Defenses

- Software bugs are difficult to detect or fix
  - Question: Can we make them harder to exploit?

- **Benign Diversity**
  - Preserve functional behavior
    - On benign inputs, diversified program behaves exactly like the original program
  - Randomize attack behavior
    - On inputs that exercise a bug, diversified program behaves differently from the original
Automated Introduction of Diversity

- Use transformations that preserve program semantics
- Challenge: how to capture intended program semantics?
  - Relying on manual specifications isn’t practical
- Solution: Instead of focusing on program-specific semantics, rely on *programming language semantics*
  - Randomize aspects of program implementation that aren’t specified in the programming language
  - Benefit: programmers don’t have to specify anything
- Examples
  - **Address Space Randomization (ASR)**
    - Randomize memory locations of code or data objects
    - Invalid and out-of-bounds pointer dereferences access unpredictable objects
  - **Data Space Randomization (DSR)**
    - Randomize low-level representation of data objects
    - Invalid copy or overwrite operations result in unpredictable data values
  - **Instruction Set Randomization (ISR)**
    - Randomize interpretation of low-level code
    - $W \oplus X$ has essentially the same effect, so ISR is not that useful any more
How randomization disrupts take-over

- Without randomization, memory errors corrupt process memory in a predictable way
  - Attacker knows what data is corrupted, e.g., return address on the stack
    - Relative address randomization (RAR) takes away this predictability
  - Attacker knows the correct value to be used for corruption, e.g., the location of injected code (in a buffer that contains data read from attacker)
    - Absolute address randomization (AAR) takes away this predictability for pointer-valued data
    - DSR takes away this predictability for all data
Corrupt non-pointer data
Compromise security critical data, e.g.,
• File names opened for write or execute
• Security credentials
  • Authenticated user?

Corrupt data pointer
• Frame pointer
• Local variables, parameters
• Pointer used to copy input

Corrupt code pointer
“Control-flow Hijack attacks”
• Return address
  • Function pointer
  • Dynamic linkage tables

Requires DSR or Relative Address Randomization

Handled by ProPolice

Pointer to injected code

Handled by Stackguard, RAD

Pointer to existing code

Handled by ISR

Corrupt a pointer value

Broken by DSR & abs. addr. randomization

Memory Error Exploits

Data Attacks

Pointer to injected data

Pointer to existing data
First Generation ASR: Absolute Address Randomization (ASLR)

- Invented by PaX project and Our Lab at SBU
- Randomizes base address of data (stack, heap, static memory) and code (libraries and executable) regions
- Implemented on many flavors of UNIX & Windows
  - UNIX implementations usually provide 20+ bits of randomness, 16 bits for Windows
- Implemented on all mainstream OS distributions
  - Linux, OpenBSD, Windows, Android, iOS, ...
- Limitations
  - Incomplete implementations (e.g., executables or some libraries left unrandomized) --- but this is becoming rare these days.
  - Brute-force attacks
  - Information leakage attacks
  - Relative address attacks
    - Non-pointer data attacks, partial pointer overwrites
Second Generation ASR: Relative Address Randomization

- Randomize distance between objects (code or data)
- [Bhatkar et al] use code transformation to permute the relative order of objects in memory
  - Static variables
  - “Unsafe” local variables
    - Safe local variables moved to a “safe” stack (no overwrites possible)
    - Safe stack option is now available on LLVM compiler
  - Heap allocations
  - Functions
  - Introduce gaps between objects
    - Some gaps may be made inaccessible
- Active current research: efficient RAR of code objects
Benefits of RAR

- Defeats the overwrite step, as well the step that uses the overwritten pointer value
  - Defeats format-string and integer overflow attacks
  - Stack-smashing attacks fail deterministically (due to safe stack)

- Higher entropy
  - Up to 28 bits on 32-bit address space
  - Knowing the location of one object does not tell you much about the locations of other objects
    - Information leakage attacks become difficult
    - Heap overflows become more difficult since you need to make two independent guesses
Data Space Randomization
DSR Technique

- **Basic idea: Randomize data representation**
  - *Xor* each data object with a *distinct random mask*
  - Effect of data corruption becomes non-deterministic, e.g.,
    - Use out-of-bounds access on array *a* to corrupt variable *x* with value *v*
      - Actual value written: $\text{mask}(a) \oplus v$
      - When *x* is read, this value is interpreted as $\text{mask}(x) \oplus (\text{mask}(a) \oplus v)$
        - Which is different from *v* as long as the masks for *x* and *a* differ.

- **Benefits**
  - Unlike AAR, protects all data, not just pointers
  - Effective against relative address as well as absolute address attacks
  - Large entropy
    - 32-bits of randomization for integers
    - Masks for different variables can be independent
      - resists information leak attacks
  - Can address intra-structure overflows
    - Not even addressed by full memory error detection techniques
DSR Transformation Approach

- For each variable \( v \), introduce another variable \( m_v \) for storing its mask

- Randomize values assigned to variables (LHS)
  - Example: \( x = 5 \) \( \rightarrow \) \( x = 5; x = x ^ { m_x} \)

- Derandomize used variables (RHS)
  - Example: \( (x + y) \) \( \rightarrow \) \( ((x ^ m_x) + (y ^ m_y)) \)

- Key problem: aliasing
  - \( \text{int } *x = \&y \)
  - A value may be assigned to \( y \) and dereferenced using \( *x \)
    - Both expressions should yield the same value
      - Need to ensure that possibly aliased objects should use the same randomization mask

- Note
  - In \( x = y \), it is not necessary to assign same mask to \( x \) and \( y \)
Summary of Automated Diversity

- Transformations that respect programming language semantics are good candidates for automated diversity
  - But they are typically good for addressing only low-level implementation errors. (We have discussed them only in the context of a specific low-level error, namely, memory corruption.)

- Automated diversity has been particularly successful in the area of memory error exploit prevention
  - First generation of randomization-based defenses focused on absolute address based attacks
    - Absolute-address randomization
    - Practical technique with low impact on systems, and hence begun to be deployed widely
  - Second generation defenses provide protection from relative-address dependent attacks
    - Relative address randomization and data-space randomization
    - Performance and compatibility (for DSR) limit widespread deployment
State of Exploit defenses and New attacks

- Most OSes now implement
  - ProPolice like defenses, plus SEH protection (Microsoft)
  - ASLR
  - DEP/NX (prevent injected code execution)

- Recent attacks
  - Exploit incomplete defenses, or use Heapspray for control-flow hijack
    - No ASLR on most executables on Linux, some EXE, DLLs on MS
    - Some libraries don’t enable stack protection, or it is incomplete
    - Heapspray: brute-force attack in the space domain
      - Exploits untrusted code in safe languages (Javascript, Java, Flash,…)
      - Code allocates almost all of memory, fills with exploit code
      - Jump to random location: with high probability, it will contain exploit code
  - Return-oriented programming (ROP) to overcome DEP
  - Rely increasingly on information leak attacks to overcome uncertainty due to ASLR, frequent software updates, and so on
    - Just-in-time-ROP: use information leak vulnerability to scan code at runtime to identify ROP gadgets
B. Preventing Memory Errors
Memory Errors in C

- **Spatial errors**: out-of-bounds subscript or pointer
  - `char *p = malloc(10); *(p+15);`

- **Temporal errors**: pointer target no longer valid
  - Uninitialized pointer
  - Dangling pointer
    - `free(p); q = malloc(...); *p;`
    - **Note**: target may be reallocated!

- **Hard to debug, especially temporal errors**
  - Unpredictable delay, unpredictable effect
    - Reallocated pointer errors are the worst kind
  - “Defensive programming” leads to memory leaks
Memory Errors in C

- **Spatial errors**: out-of-bounds subscript or pointer
  - `char *p = malloc(10); *(p+15);`

- **Temporal errors**: pointer target no longer valid
  - Uninitialized pointer
  - Dangling pointer
    - `free(p); q = malloc(...); *p;`
    - "Note: target may be reallocated!"

- **Hard to debug, especially temporal errors**
  - Unpredictable delay, unpredictable effect
    - "Reallocated pointer errors are the worst kind"
  - “Defensive programming” leads to memory leaks
Issues and Constraints

- Backward compatibility with existing C-code
  - Casts, unions, address arithmetic
  - Conversion between integers and pointers

- Compatibility with previously compiled libraries
  - Can’t expect to rebuild the entire system
  - Source code access can be problematic for some libs

- Temporal Vs Spatial Errors
  - Detecting reallocated storage
    - Important, since such errors get detected very late, and it is extremely hard to track them down

- Use of garbage collection
Why Not Garbage Collection?

- **Masks temporal errors**
  - Problematic if the intent is to use memory error-checking only during the testing phase

- **Unpredictable overheads**
  - Problematic for systems with real-time or stringent performance constraints

- **GCs can make mistakes due to free conversion between integers and pointers**
  - Fail to collect inaccessible memory
  - *Collect memory that should not be collected*
  - Problematic for code that relies heavily on such conversions, e.g., OS Kernel
Approaches for Preventing Memory Errors

- Introduce inter-object gaps, detect access to them (Red zones)
  - Detect subclass of spatial errors that involve accessing buffers just past their end
    - Purify, Light-weight bounds checking [Hasabnis et al], Address Sanitizer [Serebryany et al]
- Detect crossing of object boundaries due to pointer arithmetic
  - Detects spatial errors
  - Backwards-compatible bounds checker [Jones and Kelly 97]
  - Further compatibility improvements achieved by CRED [Ruwase et al]
  - Speed improvements: Baggy [Akritidis et al], Paricheck [Younan et al]
- Runtime metadata maintenance techniques
  - Temporal errors: pool-based allocation [Dhurjati et al], Cling [Akritidis et al]
  - Spatial + temporal errors: CMemSafe [Xu et al], SoftBounds [Nagarakatte et al]
  - Targeted approaches: Code pointer integrity [Kuznetsov et al], protects subset of pointers needed to guarantee the integrity of all code pointers.
CMemSafe: Detecting Spatial Errors Using Metadata

Spatial Check:
(p >= p_info.base &&
 p < p_info.base+p_info.size)?

char * p;
p = malloc(8);
p += 2;
*p; /* OK */
p += 14;
*p; /* error */

* base, size: base address and allocated size of the block
Temporal Check:
(*q_info.cap_ptr == VALID) ?

char * p, *q;
p = malloc(8);
q = p;
*q; /* OK */
free(p);
*q; /* error */
p = malloc(16);
*q; /* error */

- **cap_ptr**: pointer to unique capability associated with block
- **Detect erroneous accesses to freed or reallocated memory**
Credits

- Slides on Stack layout, ROP and heap overflows: courtesy Nick Nikiforakis
Vulnerabilities II: Input Validation Errors and Defenses
What comes after buffer overflows?

• Most vulnerabilities reported in the early part of 2000s were due to memory corruption
  • Typically, 2/3rd to 4/5th of security advisories
• But things have changed dramatically since then
  • Web-related vulnerabilities dominate today
    • Increased use of web
    • Hybrid nature of web applications, with server and client-side components; and a mix of trusted/untrusted data
    • Less sophisticated developers
• In the previous offering of this course, one team found 200K sites with SQL injection vulnerabilities in a few days
  • 7% of sites found using a search technique were vulnerable!
  • An even larger fraction was susceptible to cross-site scripting (XSS)
SQL Injection

- **Attacker-provided data** used in SQL queries
  
  ```
  $cmd = "SELECT price FROM products WHERE name='" . $name . "'"
  ...
  Use cmd as an SQL query
  ```

- **Attacker-provided name:**
  - `xyz'; UPDATE products SET price=0 WHERE name='iphone7s`

- **Resulting query**
  ```
  SELECT price FROM products WHERE name='xyz';
  UPDATE products SET price=0 WHERE name='iphone7s'
  ```
Command Injection

- Attacker-provided data used in creation of command that is passed to the OS
- Example: SquirrelMail

  ```php
  $send_to_list = $_GET['sendto']
  $command = "gpg -r $send_to_list 2>&1"
  popen($command)
  ```

- Attack: user fills in the following information in the “send” field of email:
  ```
  xyz@abc.com; rm -rf *
  ```
Script Injection

- Similar to command injection: attacker-provided input used to create a string that is interpreted as a script
- Common in dynamic languages since these often allow string values to be `eval`d
  - Most common web-application languages support `eval`: PHP, Python, Ruby, ...
- **Format string attacks**
  - Have similarity with script injection
    - The command language is that of format directives
Cross-Site Scripting

• Cross-Site Scripting (XSS)
  • Attacker-provided data used as scripts embedded in generated Web pages
  • Example:
    
    http://www.xyzbank.com/findATM?zip=90100

• Normal
  
    <HTML>ZIP code not found: 90100</HTML>

• Attack
  
    <HTML>ZIP code not found: <script src='http://www.attacker.com/malicious_script.js'>
    ‘http://www.attacker.com/malicious_script.js’</script></HTML>
Directory traversal

• Directory traversal
  • Attacker-provided path names contain directory traversal strings (e.g. “../”)
  • May be disguised by various encodings
  • Example:
    ```c
    void check_access(char *file) {
        if ((strstr(file, "cgi-bin/")==file) &&
            (strstr(file, "../")==NULL)) {
            char *f = url_decode(file);
            /* allow access to f ... */
    }
    ```

• Attacker-provided file:
  ```
  /cgi-bin/%2e%2e/bin/sh
  ```
Distribution of vulnerabilities: CVE 2006

- Cross-Site Scripting: 28%
- Command/Code Injection: 27%
- SQL Injection: 21%
- Format String: 6%
- Directory Traversal: 1%
- Config/Race Errors: 1%
- Memory Errors: 16%
- Other Attacks: 19%
Distribution of vulnerabilities: CVE 2009

- SQL Injection: 30%
- Cross-Site Scripting: 27%
- Memory Errors: 18%
- Config/Race Errors: 10%
- Directory Traversal: 10%
- Format String: 2%
- Other Attacks: 3%
Based on CVE reports from 2012. About half of the reports correspond to specific vulnerabilities included in this chart, the rest refer to broad classes such as “logic errors” and “weak authentication.”
A Unified View of Attacks

- **Target**: program mediating access to protected resources/services
- **Attack**: use maliciously crafted input to exert unintended control over protected resource operations
- **Resource/service access uses**:
  - Well-defined APIs to access
    - OS resources
    - Command interpreters
    - Database servers
    - Transaction servers,
    - .......
  - Internal interfaces
    - Data structures and functions within program
      - Used by program components to talk to each other

Incoming requests
(Untrusted input)

Program

Outgoing requests
(Security-sensitive operations)
Example: SquirrelMail Command Injection

- **Attack**: use maliciously crafted input to exert unintended control over output operations.
- **Detect** "exertion of control"
  - Based on **taint**: degree to which output depends on input.
- **Detect if control is intended**:
  - Requires policies
    - Application-independent policies are preferable.

---

**Incoming Request**
(Untrusted input)

- $send_to_list = $_GET['sendto']
- $command = "gpg -r $send_to_list 2>&1"
- popen($command)
- sendto="nobody; rm –rf *"

**Program**

**Outgoing Request/Response**
(Security-sensitive operations)
(To databases, backend servers, command interpreters, files, …)
Taint-Enhanced Policy Enforcement

Input Interface

Program

Fine-grained Taint Tracking

Security-Sensitive Operations

**Taint sources:** Mark untrusted data as tainted
- Marking using wrapper functions
- Usually marking network inputs as untrusted

**Fine-grained taint tracking**
- Approaches:
  - Source code transformation
  - Binary translation/emulation
  - Static analysis
- **Character-granularity taint (NOT variable granularity)**

**Taint sinks:** Enforce taint policies
- Policies as patterns on arguments of security functions
- Patterns as taint-annotated regular expressions
Instrumentation for Taint Tracking

- Fine-grained taint-tracking
  - track if each byte of memory is tainted
- Bit array `tagmap` to store taint tags of every memory byte
- `Tag(a)`: Taint bits in `tagmap` for memory bytes at address `a`

\[
\begin{align*}
x &= y + z; & \Rightarrow & \quad Tag(&x) = Tag(&y) \, || \, Tag(&z); \\
x &= *p; & \Rightarrow & \quad Tag(&x) = Tag(p);
\end{align*}
\]
Enabling Fine-Grained Taint Tracking

- Source code transformation (on C programs) to track information flow at runtime
  - Accurate tracking of taint information at byte granularity

**Idea**

- Runtime representation of taint information
  - Use bit array `tagmap` to store taint tags for each byte of memory
  - `Tag(a)`: representing taint bits of bytes at address `a` in `tagmap`

- Update `tagmap` for each assignment
## Transformation: Taint for Expressions

<table>
<thead>
<tr>
<th>$E$</th>
<th>$T(E)$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>0</td>
<td>Constants are untainted</td>
</tr>
<tr>
<td>$v$</td>
<td>$tag(&amp;v, \ sizeof(v))$</td>
<td>$tag(a, n)$ refers to $n$ bits starting at $tagmap[a]$</td>
</tr>
<tr>
<td>$&amp;E$</td>
<td>0</td>
<td>An address is always untainted</td>
</tr>
<tr>
<td>$*E$</td>
<td>$tag(E, \ sizeof(*E))$</td>
<td></td>
</tr>
<tr>
<td>$(\text{cast})E$</td>
<td>$T(E)$</td>
<td>Type casts don’t change taint.</td>
</tr>
<tr>
<td>$op(E)$</td>
<td>$T(E)$</td>
<td>for arithmetic/bit $op$</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>otherwise</td>
</tr>
<tr>
<td>$E_1 \ op \ E_2$</td>
<td>$T(E_1) \</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>otherwise</td>
</tr>
</tbody>
</table>
## Transformation: Statements

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\text{Trans}(S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = E$</td>
<td>$v = E;$</td>
</tr>
<tr>
<td></td>
<td>$\text{tag}(&amp;v, \text{sizeof}(v)) = T(E);$</td>
</tr>
<tr>
<td>$S_1; S_2$</td>
<td>$\text{Trans}(S_1); \text{Trans}(S_2)$</td>
</tr>
<tr>
<td>$\text{if } (E) S_1$</td>
<td>$\text{if } (E) \text{Trans}(S_1)$</td>
</tr>
<tr>
<td>$\text{else } S_2$</td>
<td>$\text{else Trans}(S_2)$</td>
</tr>
<tr>
<td>$\text{while } (E) S$</td>
<td>$\text{while } (E) \text{ Trans}(S)$</td>
</tr>
<tr>
<td>$\text{return } E$</td>
<td>$\text{return } (E, T(E))$</td>
</tr>
<tr>
<td>$f(a) { S }$</td>
<td>$f(a, ta) {\text{tag}(&amp;a, \text{sizeof}(a)) = ta; \text{Trans}(S)}$</td>
</tr>
<tr>
<td>$v = f(E)$</td>
<td>$(v, \text{tag}(&amp;v, \text{sizeof}(v))) = f(E, T(E))$</td>
</tr>
<tr>
<td>$v = (*f)(E)$</td>
<td>$(v, \text{tag}(&amp;v, \text{sizeof}(v))) = (*f)(E, T(E))$</td>
</tr>
</tbody>
</table>
Implicit flows

• (Positive) control dependence
  • Example: decoding using `if-then-else/switch`
    ```
    if (x == ‘+’) y = ‘ ’;
    ```
• Negative control dependence
  ```
  y = 1;
  if (x == 0)
    y = 0
  ```
  • If x is tainted, but equals 1, then is y tainted at the end?
• Operations involving tainted pointers
  ```
  char transtab[256];
  ...
  x = transtab[p]
  ```
  • If p is tainted, is x tainted?
  • What about the following case:
    ```
    *p = ‘a’
    ```
  • Or the case:
    ```
    x = hash_table_lookup(p)
    ```
Issues in Taint-tracking Instrumentation

• **Efficiency**
  • Almost every statement is instrumented
  • Compounded when dealing with binaries
    • Can introduce 4x to 40x slowdown!

• **Accuracy**
  • Implicit flows
    • Full implicit flow support leads to far too many false positive
      • It is necessary to be very selective in terms of which implicit flows are taken into account.
    • Malicious code can disguise all flows in implicit flows, making it infeasible to do accurate taint-tracking

• Untransformed libraries
Application-independent policies

- **Lexical confinement**
  - Ensure that tainted data does not cross a word boundary
  - For binary data, can interpret struct fields as words
    - Or more coarsely, activation records or heap blocks

- **Syntactic confinement (more relaxed)**
  - Tainted data should not begin in the middle of one subtree of the parse tree and “overflow” out of it
Symlink attacks

- Do not assume that symlinks are trustworthy:
  - Example 1
    - Application A creates a file for writing in /tmp. It assumes that since the file name is unusual, or because it encodes A's name or pid, there is no need to check if the file is already present.
    - Attacker creates a symlink with same name that points to an important file F. When root runs A, F will be overwritten.
  - Example 2
    - User A runs an application that creates a file in /tmp/x and then later updates it.
    - User B attacks this application by removing /tmp/x and then creating a symlink named /tmp/x that points to an important file F.
- Hard links and file/directory renames can also be used to carry out some of these attacks, but they are difficult because there are more restrictions on them.
Race conditions

- Time-of-check-to-time-of-use (TOCTTOU) attacks
  - Often arise when an application tries to protect itself against name-based attacks

- Example
  - A setuid application permits a non-root user to specify the name of an output file, say, for logging
  - It checks if the real user has permission to write this file, usually using the access system call
  - Attacker modifies the file between access and open
    - Checks OK, but the attack succeeds!
Race condition examples

- access/open
- chmod/chown
- Directory renames
  - Root invokes `rm -r` on `/tmp/*` to clean up `/tmp`
  - Attacker creates a directory `/tmp/a` and then another directory `/tmp/a/b`
  - `rm` may (1) `cd` into `/tmp/a/b`, remove all files in it, (2) `cd` into “..”, (3) continue to remove files in `/tmp/a`, (4) `cd` “..” and (5) continue to remove files in `/tmp`
  - Attacker moves `/tmp/a/b` to `/tmp` between (1) and (3), causing files in `/` to be removed in step (5).
Succeeding in Races ...

- It may seem that it would be hard for the attacker to succeed, but he can mount “algorithmic complexity attacks”
  - Make a normally fast operation take very long
  - Example: Instead of creating a file /tmp/a, make it point to a symlink which in turn points to a symlink and so on. Access operation, which needs to resolve this sequence of symlinks will take very long. Can further slow it down by creating deep directory trees.
  - As a result, races can succeed with near 100% probability!
Avoiding filename related pitfalls

- When creating new files, call open with appropriate flags to ensure creation of new file
  - On UNIX, O_CREAT and O_EXCL flags
- Use OS-provided functions to create temp files
  - On UNIX, use mkstemp or tmpfile, not tmpnam
- Use most restrictive permission applicable
  - Always restrict writes to owners, and if possible, reads too.
  - If possible, first create a directory that is accessible only to the owner, and operate within this directory
- Configure shared directory permissions correctly
  - Use the sticky bit
Common Software Vulnerabilities

- CWE (Common Weakness Enumeration) is an excellent source on currently prevalent software vulnerabilities

- CWE Top-25 is a good point to start
  - You are expected to be familiar with the vulnerabilities in this list – read the list and understand what each vulnerability means
Common Software Weaknesses

- Input validation
  - Injection vulnerabilities
    - Cross-site scripting, SQL/command injection, code/script injection, format-string, path-traversal, open redirect, ...
  - Buffer overflows
    - Integer overflows, incorrect buffer size or bounds calculation
  - Many other application-specific effects of untrusted input

- Failure to recognize or enforce trust boundaries
  - Calling function that trust their inputs with untrusted data
  - Including code without understanding its dependencies
  - Relying on form data or cookies in a web application

- Missing security operation
  - Authentication: missing, weak, or using hard-coded credentials
  - Authorization: missing checks
    - Cross-site request forgery
  - Failure to encrypt, hash, use salt, ...
Common Software Weaknesses

- Use of weak security primitives
  - Weak random numbers, encryption, hash algorithms, ...

- Information leakage
  - Error messages that reveal too much information
    - Software version, source code fragments, database table names or errors, ...
    - Timing channels

- Execution with unnecessary privileges
  - Executing code with admin privileges
  - Incorrect (or missing) permission settings

- Error/exception-handling code
  - Failure to check error codes, e.g., open, malloc, ...
  - Failure to test error/exception-handling code

- Race conditions
Other References for Vulnerabilities

- **CWE-1000: Research view of CWEs**
  - Top 25 is useful to understand current trends, but the descriptions can often be uninformative
  - CWE-1000 organization has a much better structure and organization
  - You don’t necessarily get a sense of completeness from these, but reading them will still significantly broaden your understanding of software vulnerabilities and more secure coding practices.

- **Common Attack Pattern Enumeration/Classification**
  - From the perspective of how attacks work
  - Geared to identify principal features of these attacks
Secure Coding Practices

- The goal of this course is to expose you to a range of vulnerabilities and exploits, so you can learn how to build secure systems and develop secure code.
- But we don’t necessarily provide a “cook book”
  - The hope is that you will learn more from understanding the examples in depth than reading a long laundry list.
- Nevertheless, several good sources are available on the Internet that discuss secure coding practices:
  - CERT top 10 secure coding practices
  - CERT Secure coding standards for C, C++, and Java
  - OWASP Secure coding principles
Principles of Secure System Design

- [Saltzer and Shroeder 1975]
- Principles of
  - Economy of mechanism (simplicity => assurance)
  - Fail-safe defaults (default deny)
  - Complete mediation (look out for ways in which an access control mechanism may be bypassed)
  - Open design (no security by obscurity)
  - Separation of privilege (similar to separation of duty)
  - Least privilege
  - Least common mechanism (avoid unnecessary sharing)
  - Psychological acceptability (onerous security requirements will be actively subverted by users)
Principles of Secure System Design

- Two principles mentioned, but not recommended in [Saltzer and Shroeder 1975]
  - Work factor: how much effort will it take to break a mechanisms, versus potential gain for the attacker
    - Difficult to estimate cost
    - Sometimes, difficult to estimate gain
  - Compromise recording (maintain adequate audit trail)
    - Difficult to ensure integrity of audit records maintained on a protected system
      - These records can be compromised if stored on protected system
      - Can work if audit trail can be protected, e.g., off-site storage, tamper-proof storage systems
Vulnerabilities Vs Malicious Code

• These two pose very different threats
  • With vulnerable code, you have a relatively weak adversary: one that is constrained to exploiting an existing vulnerability, but has no way of controlling it.
  • So, relatively weak defenses such as randomization can be attempted.
  • With malicious code, you have a strong adversary
    • Can modify code to evade specific defenses
    • You cannot make assumptions such as the absence of intentionally introduced errors, obfuscation, etc.
Static and Dynamic Analysis for Vulnerability Detection
Vulnerability Analysis

- Programmer checks the program, and corrects the errors
- Cycle repeated until all relevant bugs are fixed
**Terminology**

- **False Positives:** A warning or error is generated, but there is no real vulnerability.

- **False Negatives:** A vulnerability exists, but it is not being identified by the analysis.

- **Complete:** A technique that is guaranteed to be free of false positives.

- **Sound:** A technique that is guaranteed to detect all vulnerabilities (i.e., no FNs).

- **Note:** A technique cannot be sound and complete, since most program properties are undecidable in general.

- **Useful bug-finding tools suffer from both FNs and FPs**
Benefits and Drawbacks

❖ Benefits
  ▪ Does not rely on bugs being exercised: fix the bug before it strikes you
  ▪ No runtime overhead
  ▪ Leverage programmer knowledge

❖ Drawbacks
  ▪ Not applicable for operator use
    ▼ May not have source code access
    ▼ May not be able to understand the logic of the program
  ▪ Suffers from false positives
    ▼ A programmer can cope with these, but not an operator
Vulnerability Analysis Techniques

- **Static analysis**
  - Analysis performed before a program starts execution
  - Works mainly on source code
    - Binary static analysis techniques are rather limited
  - Not very effective in practice, so we won’t discuss in depth

- **Dynamic analysis**
  - Analysis performed by executing the program
  - Key challenge: How to generate input for execution?
  - Two main approaches to overcome challenge
    - Fuzzing: random, black-box testing (primarily)
    - Symbolic execution: systematic technique for generating inputs that exercise “interesting program paths.”
      - More of a white-box approach.
Black-box fuzzing

**BlackBoxFuzzing**
Input: initial test suite $TestSuite$
Output: bug triggering inputs $Crashers$

**Mutations** (helper function)
Input: test input $t$
Output: new test inputs with some bits flipped in $t$

while $TestSuite$ not empty:
    $t = \text{PickFrom}(TestSuite)$
    for each $m$ in Mutations($t$):
        $\text{RunAndCheck}(m)$
        if Crashes($m$):
            add $m$ to $Crashers$

**Drawbacks**
- Blind search: a successful mutation does not help subsequent search in any way
Coverage guided fuzzing

CoverageGuidedFuzzing
Input: initial test suite TestSuite
Output: bug triggering inputs Crashers

while TestSuite not empty:
  t = PickFrom(TestSuite)
  for each m in Mutations(t):
    RunAndCheck(m)
    if Crashes(m):
      add m to Crashers
    if NewCoverage(m)
      add m to TestSuite

Note: A successful mutation feeds into other mutations.

AFL Fuzzer by Zalewski ‘14
Coverage metrics

❖ **Statement coverage**
   - A statement is *covered* by a test input if it is executed at least once by the program when processing that input

❖ **Edge (or branch) coverage**
   - An edge is covered by a test input if it is taken at least once when processing that input

❖ **Counted coverage**
   - Take into account the number of times a statement (or edge) is executed. Variant: use $\log\text{(count)}$ instead of exact count.

❖ **Path coverage**
   - Similar, but applies to a full execution path
   - Note: number of possible execution paths can be extremely large, or even be infinite, so it is not used.
Coverage metric

```c
void path_explosion(char *input) {
    int count = 0;
    for (int i = 0; i < 100; i++)
        if (input[i] == 'A')
            count++;
}
```
int walk_maze(char *steps) {
...
    int x, y;  // Player position.
    for (int i = 0; i < ITERS; i++) {
        switch (steps[i]) {
            case 'U': y--; break;
            case 'D': y++; break;
            case 'L': x--; break;
            case 'R': x++; break;
            default: // Wrong command, lose.
        }
        if (maze[y][x] != ' ') // Lose.
            if (maze[y][x] == '#') // Win!
                ...}

Winning input: DDDDRRRRUULUUURRRRDDDDRRUUUUU
AFL – state-of-the-art fuzzing

process timing
- run time: 0 days, 0 hrs, 4 min, 43 sec
- last new path: 0 days, 0 hrs, 0 min, 26 sec
- last uniq crash: none seen yet
- last uniq hang: 0 days, 0 hrs, 1 min, 51 sec

overall results
- cycles done: 0
- total paths: 195
- uniq crashes: 0
- uniq hangs: 1

cycle progress
- now processing: 38 (19.49%)
- paths timed out: 0 (0.00%)

map coverage
- map density: 1217 (7.43%)
- count coverage: 2.55 bits/tuple

findings in depth
- favored paths: 128 (65.64%)
- new edges on: 85 (43.59%)
- total crashes: 0 (0 unique)
- total hangs: 1 (1 unique)

stage progress
- now trying: interest 32/8
- stage execs: 0/9990 (0.00%)
- total execs: 654k
- exec speed: 2306/sec

path geometry
- levels: 3
- pending: 178
- pend fav: 114
- imported: 0
- variable: 0
- latent: 0

fuzzing strategy yields
- bit flips: 88/14.4k, 6/14.4k, 6/14.4k
- byte flips: 0/1804, 0/1786, 1/1750
- arithmetics: 31/126k, 3/45.6k, 1/17.8k
- known ints: 1/15.8k, 4/65.8k, 6/78.2k
- havoc: 34/254k, 0/0
- trim: 2876 B/931 (61.45% gain)
Bugs found by AFL

IJK jpeg 1, libjpeg-turbo 1 2, libpng 1, libtiff 1 2 3 4 5, mozjpeg 1, PHP 1 2 3 4 5, Mozilla Firefox 1 2 3 4, Internet Explorer 1 2 3 4, Apple Safari 1, Adobe Flash / PCRE 1 2 3 4, sqlite 1 2 3 4, OpenSSL 1 2 3 4 5 6 7, LibreOffice 1 2 3 4, poppler 1, freetype 1 2, GnuTLS 1, GnuPG 1 2 3 4, OpenSSH 1 2 3, PuTTY 1 2, ntpd 1, nginx 1 2 3, bash (post-Shellshock) 1 2, tcpdump 1 2 3 4 5 6 7 8 9, JavaScriptCore 1 2 3 4, pdfium 1 2, ffmpeg 1 2 3 4 5, libmatroska 1, libarchive 1 2 3 4 5 6 7, wireshark 1 2 3, ImageMagick 1 2 3 4 5 6 7 8 9, BIND 1 2 3, QEMU 1 2, Icms 1, Oracle BerkeleyDB 1 2, Android / libstagefright 1 2, iOS / ImageIO 1, FLAC audio library 1 2, libsnprintf 1 2 3 4, less / lesspipe 1 2 3, strings (+ related tools) 1 2 3 4 5 6 7, file 1 2 3 4, dpkg 1 2, rcs 1, systemd-resolved 1 2, libyaml 1, Info-Zip unzip 1 2, libtasn1 1 2, OpenBSD pfctl 1, NetBSD bpf 1, man & mandoc 1 2 3 4 5, ... , IDA Pro [reported by authors], clamav 1 2 3 4 5, libxml2 1 2 4 5 6 7 8 9, glibc 1, clang / llvm 1 2 3 4 5 6 7 8 9, nasm 1 2, ctags 1, mutt 1, procmail 1, fontconfig 1, pdksh 1 2, Qt 1, wavpack 1, redis / lua-cmsgpack 1, taglib 1 2 3, privoxy 1 2 3, perl 1 2 3 4 5 6 7 8 9, libxmp, radare2 1 2, SleuthKit 1, fwnknp [reported by author], X.Org 1 2, exifprobe 1, jhead [?] ... , capnproto 1, Xerces-C 1 2 3, metacam 1, djvulibre 1, exiv 1, Linux btrfs 1 2 3 4 5 6 7 8, Knot DNS 1, curl 1 2, wpa_supplicant 1, libde265 [reported by author], dnsmasq 1, libbpg 1 2, lame 1, libwmf 1, uudecode 1, MuPDF 1, imlib2 1, libraw 1, libbson 1, libsass 1, yara 1 2 3 4, W3C tidy-html5 1, VLC 1, FreeBSD syscons 1 2 3, John the Ripper 1 2, screen 1 2 3, tmux 1 2, mosh 1, UPX 1, indent 1, openjpeg 1, MMIX 1, OpenMPT 1 2, rxvt 1 2, dhcpcd 1, Mozilla NSS 1, Nettle 1, mbed TLS 1, Linux netlink 1, Linux ext4 1, Linux xfs 1, botan 1, expat 1 2, Adobe Reader 1, libav 1, libical 1, OpenBSD kernel 1, collectd 1, libidn 1 2
JPEGs out of thin air
Fuzzing: strength and weaknesses

if (input == 0x1badc0de) {
    ...
}

if (adler32(input) == 0x3eb52a45) {
    ...
}
Dynamic symbolic execution (DSE)

DynamicSymbolicExecution
Input: initial test suite TestSuite
Output: bug triggering inputs Crashers

while TestSuite not empty:
    t = PickFrom(TestSuite)
    for each m in DSENewInputs(t):
        RunAndCheck(m)
        if Crashes(m):
            add m to Crashers
        add m to TestSuite
Dynamic symbolic execution

**DSENewInputs**

Input: test case $t$
Output: new test cases $Children$

$PC = \text{ExecuteSymbolically}(t)$
for each condition $c$ in $PC$:

NEW $PC = PC[0..i-1]$ and not $c$

$\text{new\_input} = \text{SMTSolve}(\text{NEW\_PC})$

if $\text{new\_input} \neq \text{UNSAT}$:

add $\text{new\_input}$ to $Children$
Constraint solvers (SAT/SMT)

• Complete solvers (most used)
  • Complete = always returns an answer (*given enough time*)
  • **Backtracking** based algorithms
  • Typically based on Conflict-Driven Clause Learning (CDCL) algorithm
  • E.g., **Z3** from Microsoft Research, STP (used by KLEE)

• Incomplete solvers
  • Incomplete = may return “don’t know”
  • Trade-off between complexity and the quality of the search
  • **Stochastic local search (SLS)** based algorithms
  • E.g., **SLS-SMT** by Frohlich et al. [AAAI’15]

• Note that theoretically complete solvers are indeed incomplete in their practical use, since implementations call the solver, and time out after a specific period.
Fuzzing vs. DSE

<table>
<thead>
<tr>
<th>Technique</th>
<th>Replayable</th>
<th>Semantic Insight</th>
<th>Scalability</th>
<th>Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Symbolic Execution</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
<td>16</td>
</tr>
<tr>
<td>Veritestng</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>11</td>
</tr>
<tr>
<td>Dynamic Symbolic Execution + Veritestng</td>
<td>Yes</td>
<td>High</td>
<td>Medium</td>
<td>23</td>
</tr>
<tr>
<td>Fuzzing (AFL)</td>
<td>Yes</td>
<td>Low</td>
<td>High</td>
<td>68</td>
</tr>
</tbody>
</table>

“In reality, fuzzing identified almost three times as many vulnerabilities [as DSE]. In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution.” ANGR Study (The Art of War) [Oakland’16]
DSE: strength and weaknesses

- Symbolic state maintenance is costly
  - Overhead of executing symbolically can be \( \sim 1000x \) [SAGE]
- Constraint solving does not scale well (NP-hard problem)
  - Time complexity: complex formulas often time out
- Path condition solved by the solver is not guaranteed to take the targeted path
  - Due to imperfections of the symbolic memory model and environment model
  - Path divergence in 60% of the case [SAGE]
- The probability of a new test case exercising a new path is still much higher than in case of blind fuzzing
Binary Instrumentation

R. Sekar
Stony Brook University
Limitations of SFI/NaCl Approach

♦ Need for compiler support
  ◆ Does not work on arbitrary binaries --- binaries should have been compiled using a cooperative compiler
  ◆ Otherwise, the binary will trivially fail the verification step

♦ Question: Can we instrument arbitrary COTS binaries to insert inline security checks?
Motivation for COTS Binary Instrumentation

- No source code needed
  - Language-neutral (C, C++ or other)
- Can be largely independent of OS
- Ideally, would provide instruction-set independent abstractions
  - This ideal is far from today’s reality

Benefits
- Application extension
  - Functionality
  - Security
  - Monitoring and debugging
- Instrumenting long-running programs
Approaches

- **Static analysis/transformation**
  - Binaries files are analyzed/transformed
  - **Benefits**
    * No runtime performance impact
    * No need for runtime infrastructure
  - **Weakness**
    * Error-prone, problem with signed code (can work around)

- **Dynamic analysis/transformation**
  - Code analyzed/transformed at runtime
  - **Benefit**: more robust/accurate
  - **Weakness**
    * High runtime overhead
    * Runtime complexity (infrastructure)
Previous Works (Static)

- OM/ATOM (DEC WRL)
  - Proprietary and probably outdated

- EEL (Jim Larus et al, 1995)
  - The precursor of most modern rewriters
  - Targets RISC (SPARC)
  - Provides processor independent abstractions

- Follow up works
  - UQBT (for RISC)
  - LEEL (for Linux/i386)
Previous Works (Dynamic)

- **LibVerify (Bell Labs/RST Corp)**
  - Runtime rewriting for StackGuard

- **DynamoRIO (HP Labs/MIT)** Disassembles basic blocks at runtime
  - Provides API to hook into this process and transform executable

- **Pin (Intel/U. Colorado)**

- **Valgrind**

- A number of virtualization implementations rely binary translation (or used to)
  - QEMU, VMWare, ...
Phases in Static Analysis of Binaries

- Disassembly
- Instruction decoding/understanding
- Insertion of new code
Static Disassembly

- Core component for static analysis of binaries

- Principal Approaches
  - Linear Sweep
  - Recursive Traversal
Linear Sweep Algorithm

- Used by GNU `objdump`

**Problem:**
- There can be data embedded within code
  - There may also be padding, alignment bytes or junk
- Linear sweep will incorrectly disassemble such data

```c
Addr = startAddr;
while (Addr < endAddr ) {
    ins=decode(addr);
    addr+=LengthOf(ins);
}
```
Linear Sweep Algorithm

- 804964c: 55 push %ebp
- 804964d: 89 e5 mov %esp,%ebp
- 804964f: 53 push %ebx
- 8049650: 83 ec 04 sub $0x4,%esp
- 8049653: eb 04 jmp 0x8049658
- 8049655: e6 02 04 <junk>
- 8049658: be 05000000 mov $0x5,%esi

- 804964c: 55 push %ebp
- 804964d: 89 e5 mov %esp,%ebp
- 804964f: 53 push %ebx
- 8049650: 83 ec 04 sub $0x4,%esp
- 8049653: eb 04 jmp 0x8049658
- 8049655: e6 02 out 0x2, al
- 8049657: 04 be add al, 0xbe
- 8049659: 05 00000012 add eax, 0x12000000

Incorrectly disassembles junk (or padding) bytes
Confusion typically cascades past the padding, causing subsequent instructions to be missed or misinterpreted.
Self Repairing Disassembly

- Property of a disassembler where it re-synchronizes with the actual instruction stream
- Makes detecting disassembly errors difficult
  - 216 of 256 opcodes are valid
- Observation: re-synchronization happens quickly, within 2-3 instructions beyond point of error.
Self Repairing Disassembly (example)

Consider the byte stream
55 89 e5 eb 03 90 90 83 0c 03 b8 01 00 00 00 c9

- **Linear Sweep output**
  100: push ebp
  101: mov ebp, esp
  103: jmp 109
  105: nop
  106: nop
  107: or dword ptr ds:[ebx+eax*1], 0xb8
  111: add dword ptr ds:[eax+eax*1], eax
  113: add byte ptr ds:[eax+eax*1], al
  116: leave

- **Correct Output**
  100: push ebp
  101: mov ebp, esp
  103: jmp 109
  106: <GAP>
  107: <GAP>
  108: <GAP>
  109: or al, 0x3
  111: mov eax, 0x1
  116: leave
Recursive Traversal

- **Approach**: Takes into account the control flow behavior of the program.
- **Weakness**: For indirect jumps, jump target cannot be determined statically, so no recursive traversal of the target can be initiated.
- **Some error cases not handled**, e.g., jump to the middle of an instruction.

```c
RecursiveTraversal (addr) {
    while (!visited[addr]) {
        visited[addr] = true;
        ins = decode (addr);
        if (isControlTransfer(ins))
            RecursiveTraversal (target(ins))
        if (uncondJumpOrRet(ins))
            return
        else addr+=LengthOf(ins);
    }
}
```
Static Disassembly - Impediments

- Code/Data distinction
- Variable x86 instruction size

Indirect Branches

- Mainly due to function pointers
  - Cross-module calls
    - e.g., calls from executable to a library
  - GUI code (event handlers)
  - C++ code (Virtual functions)
- Functions without explicit CALL

- PIC (Position-Independent Code)
- Hand-coded Assembly
#include <stdio.h>

void f(int c) {
    printf("%d\n", c);
}

void h(int i) {
    f(i+1);
}

int i(int j) {
    return j+1;
}

int main(int argc, char*argv[]) {
    h(i(argc));
    f(argc+2);
}
Compiled Code Example

```c
void f(int c) {
    printf("%d\n", c);
}
```

Function prologue

```assembly
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
pushl   8(%ebp)
pushl   $.LC0 ("%d")
call    printf
leave
ret
```
Optimized Code Example

```c
void h(int i) {
    f(i+1);
}
```

No push of arguments

```
pushl    %ebp
movl    %esp, %ebp
subl    $8, %esp
incl    8(%ebp)
leave
jmp     f
```
Optimized Code Example

```c
main(int argc, char*argv[]) {
    h(i(argc));
    f(argc+2);
}
```

Return value in eax reg,
No argument push!

No push of arguments to f, tail call
Static Code Transformation Limitations

- Cannot move code
  - Cannot predict the destination of indirect calls, so there is no safe way to move code
  - Can copy code, if original is left in place
    - If the goal is to protect against vulnerabilities in original code, then leaving of original code defeats this purpose!

- Code insertion is tricky
  - Obvious approach: overwrite original code with unconditional jump, patch
  - Problem: There may not be enough room for jump instruction (5 bytes)
  - Possible solution: use INT 3 (one-byte) instruction
    - Higher overhead for handling (signal generation/handling)
Static Code Transformation Limitations

- Code insertion at arbitrary points is very difficult
  - Code insertion at beginning/end of function is easy
  - Other points in code are not well defined in optimized code
    - Loops may be unrolled
    - Switch statements translated to jump tables
    - Successive branches may be combined
    - Function arguments may not be explicitly pushed (nor return value popped)
    - Tail call optimization and function inlining
Code Transformation Limitations

- Relocation of static data is not feasible
  - Cannot identify and relocate static pointers, which appear as immediate constants in assembly code
  - Note: These constants may be passed through several functions before used

- Note: Most above limitations can be removed if relocation information is present in binary

- Relocation of stack/heap data possible
  - Change SP at program beginning
  - Intercept/modify malloc and/or mmap requests
Dynamic Transformation Techniques
Libverify

- Inserts (StackShield) checks in binary code
  - Copy each function to heap
  - Modify first and last instruction in each function to jump to wrapper code implementing StackShield
  - Replace original copy with TRAP instructions
    * Any jump/calls to original code activates trap handler
    * Handler looks up corresponding address in copied code and transfers control there
    * If no copy exists (entry to function not discovered at load time), the function can be copied to heap and instrumented at this time

- Benefit: handles all indirect control transfers correctly
  - Note: indirect transfers will go to original code locations, unless code pointers are identified and modified to point to new locations
    * As mentioned before, this is hard to do in many cases, so this approach of using traps and runtime redirection is a good trade-off between performance and compatibility
  - Drawback: performance impact if traps are repeatedly executed
Use Libverify’s runtime handling to the extreme

- All code is discovered dynamically, analyzed dynamically, and then rewritten
- Code is transformed one basic block at a time
  - Side-steps the thorny problem of disassembly
  - Note that it is trivial to reliably disassemble a single basic block, which is straight-line code with no control-transfers in the middle.
- Only the first execution of a basic block requires analysis and rewriting. Subsequent executions can use the same rewritten block.

Control transfers occur in the last instruction of a basic block. These instructions need to be checked at runtime.

Non-control-transfer instructions are executed natively
DynamoRIO Operation

⚠️ Instrumented programs run in two contexts

♦ DynamoRio context (above the redline, representing DynamoRIO runtime). Responsible for detecting the execution of new basic blocks (BBs)
  - These BBs are disassembled, analyzed and then transformed: just-in-time disassembly/rewriting, just before first execution
  - DynamoRIO provides an API for instrumentation: one can use this API to implement custom instrumentation, e.g., count number of BBs executed, number of memory accesses, etc.

♦ Application context (below the red line, application code executes natively)
  - Non-control-transfer instructions need no special treatment
  - Control-transfers need to be checked
    ♦ If they are direct transfers, then we check if the target has already been instrumented (and hence is in the code cache). If so, directly jump there. If not, switch into DynamoRIO context to perform instrumentation.
    ♦ Indirect transfers need to go through a translation table
Handling Indirect CFT

▲ Note that indirect control transfers will use original code addresses

▲ But the instrumented code is in the code cache at a different address. (We cannot use the original addresses, even if they were available: instrumentation causes code to expand, so every target except the very first function in the instrumented application will necessarily reside at a different location as compared to the original code.)

▲ As discussed before, we cannot “fixup” code references either. This is because code addresses will be immediate constants in the binary, and there is no way to distinguish integer constants from code addresses
  * If we mistakenly “fixup” an integer value, that will change program behavior
  * If we mistakenly omit the fixup of a code pointer, then code will jump to an incorrect location, likely leading to a crash

▲ Clever idea put forth by DynamoRIO authors

▲ Wait until a pointer is actually used
  * If it is used as a target of control transfer, then it is obviously a code pointer
  * Just-in-time code pointer fixup: fixup happens at the very last step.
Fixup Implementation

Fixup is implemented using a translation table

- A hash table jmptab maps the original address of a BB to its new address in the code cache (corresponding to the location of the instrumented version of code)
- Each time DynamoRIO runtime instruments a BB, it enters the mapping between the original location and the new location in this table.

At runtime, every indirect CFT to a location \( l \) is translated into \( \text{jmptab}[l] \)

- Each indirect jump requires a hash table lookup, and has a performance cost
- Fortunately, common cases (e.g., returns and repeated calls to same target) can be optimized

If the target is not in \( \text{jmptab} \), then control transferred to DynamoRIO runtime.
DynamoRIO Context Switch

- Preserve the following conditions
  - All GPRs (8 in x86-32)
  - Eflags
  - Some system state. Eg: error code

  - DynamoRIO uses one slot in TLS (thread local storage) to store error code (errno) of the application.
  - DynamoRIO will use some library routines that may modify the state as error code, so it is necessary to preserve application’s errno.
DynamoRIO Context Switch

Assumes that the BBs at 0x40106f and the immediately following BBs are not in the code cache. In this case, control has to be transferred to DynamoRIO runtime when execution reaches the end of this BB. Before context switch, all of the application state (in particular, registers) need to be saved.
Transparency & OS Issues

• Transparency: application cannot tell that it is running inside DynamoRIO

• Why does DynamoRIO need transparency?
  – Ensures that application behaves exactly the same way as before: it can’t even tell the difference.
  – So, it can’t evade DynamoRIO, nor can it behave differently.

• Transparency Issues
  – Library transparency
  – Thread transparency
  – Stack transparency
  – Address space transparency
  – Context Translation
  – Performance transparency (not preserved)
Library Transparency

• **Issues** when both DynamoRIO and application enters the **same non-re-entrant library routine**
  – System state might be broken (errno)
  – Library routine may fail to work (malloc)

• **Solution:**
  – Use **system call** on both windows and Linux
  – Use **stateless library routines**
  – Implement own **memory (de)allocation routines**.
Thread Transparency

• DynamoRIO does not create its own thread

• Why?
  – violate transparency when application that monitors all reads in a process
  – Performance issue when threads double

• What about one DynamoRIO thread?
  – Still violate transparency
  – Performance degrades when multiple threads switch into DynamoRIO mode

• Therefore, use app thread with new context
Stack Transparency

• DynamoRIO does not “touch” application stack.
  – Some applications may access data beyond the top of stack. Eg: Microsoft office
  – Usual stack conventions may not be followed by hand-crafted assembly
    – use of esp as a GPR
  – Ability to read return address off stack and use in computing code location (or modify it)
    – Used in PIC (position-independent code)

• Solution:
  – Use a private stack for each thread in DynamoRIO mode
  – Do not modify content of original stack
Address Space Transparency

• DynamoRIO should not “leak” information about itself.
  – On Windows, intercept
    • NtQueryVirtualMemory() that traverse memory regions
    • GetModuleFileName() (library call) to check if library is present
  – On Linux, intercept
    • mmap(). etc.

• More measures (security)
  – Mark DynamoRIO code as NX, when in code cache
Context Translation

• When exception occurs, the faulting place should be the original code address.
  – Intercept user signal handler
  – Check the address map, find the original address
  – Modify the signal stack and go to user signal handler
Transparency & OS Issues

• Operating System Issues
  – Kernel Mediated Control Flow
  – System Call Handling
  – Thread synchronization
Kernel Mediated Control Flow

- Signal Handling
  - DynamoRIO routine will get control first
  - Signals will be queued and delayed, except urgent signals
    - Eg: SIGSEGV

- When signal arrives, if the thread is at
  - Code cache:
    - Unlink the current basic block, go back to DynamoRIO
    - If bb contains syscall, jump to exit stub before syscall.
      - Why? Bound timing of signal handler, since syscall is expensive.

- DynamoRIO code:
  - Delay signal until reaching a safe place
  - Emulate kernel behavior
System Call handling

- If syscall number is not statically known or on DynamoRIOs list
  - Insert pre-syscall & post-syscall routines around the instruction
- Uninterested syscall: left unchanged. However:
- For signal handling, app must LEAVE code cache QUICKLY (for timing issue)
  - Insert a jump prior to the syscall:
    - Jmp <syscall or bail>
    - Bail: jmp <exit stub>
    - Syscall:
    - <system call instruction>
Program Shepherding: An IRM based on DynamoRIO

- Introduces in-line checks to defend against common exploits
  - Buffer overflow attacks
  - Format string attacks
  - Injection of malicious code
  - Re-use of existing code (existing code attacks)
- Sandboxing
gcc is slow since it consists of many short runs with little code re-use.

Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN 90 benchmarks) on Linux. They were compiled using gcc -O3. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.
Windows is much less efficient at changing privileges on memory pages than Linux.
Caveat about performance

- DBT performance measurements usually based very long-running CPU-intensive benchmarks
- These applications represent the “best case scenario” for DBT systems
  - Rewrite once, execute for a long time
- Real-world performance can be bad
  - 10x to 40x slowdown in the worst case
- Example DBT systems
  - DynamoRIO, Pin, Valgrind, ...
- But its exceptional level of compatibility with arbitrary binary code can still be compelling for
  - CPU-intensive applications with tight loops
  - Coarse-granularity instrumentation (i.e., very small fraction of instructions instrumented)
  - Debugging applications
Other Dynamic Transformation Tools

- **Pin**
  - better supported now than DynamoRIO
  - better engineered for Linux

- **Strata**

- **Valgrind**
  - Most popular open-source tool for finding memory errors and many other applications

- **Qemu**
  - Can support whole system emulation
DynamoRIO vs Pin

• Architecture dependency
  – Pin tools: written in c/c++
  – DynamoRIO: written in x86 assembly
  – DynamoRIO’s tools allow users to operate at a lower level
    – Have more control over efficiency, but programming can be hard, and architecture dependent.
BBCount Pin Tool

For more information, including tutorials and examples, see https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

```c
static int bbcount;

VOID PIN_FAST_ANALYSIS_CALL docount() { bbcount++; }

VOID Trace(TRACE trace, VOID *v) {
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
        bbl = BBL_Next(bbl)) {
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, AFUNPTR(docount),
                        IARG_FAST_ANALYSIS_CALL, IARG_END);
    }
}

int main(int argc, char *argv[]) {
    PIN_InitSymbols();
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_StartProgram();
    return 0;
}
```
static int global_count;

static dr_emit_flags_t event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
    bool for_trace, bool translating) {
    instr_t *instr, *first = instrlist_first(bb);
    uint flags;
    /* Our inc can go anywhere, so find a spot where flags are dead. */
    for (instr = first; instr != NULL; instr = instr_get_next(instr)) {
        flags = instr_get_arith_flags(instr);
        /* OP_inc doesn't write CF but not worth distinguishing */
        if (TESTALL(EFLAGS_WRITE_6, flags) && !TESTANY(EFLAGS_READ_6, flags))
            break;
    }
    if (instr == NULL)
        dr_save_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    instrlist_meta_preinsert(bb, (instr == NULL) ? first : instr,
        INSTR_CREATE_inc(drcontext, OPND_CREATE_ABSMEM((byte *)&global_count, OPSZ_4)));
    if (instr == NULL)
        dr_restore_arith_flags(drcontext, bb, first, SPILL_SLOT_1);
    return DR_EMIT_DEFAULT;
}

DR_EXPORT void dr_init(client_id_t id) {
    dr_register_bb_event(event_basic_block);
}
Applicability of Static Vs Dynamic Techniques

- Some techniques require static instrumentation
  - Any technique that uses static analysis to compute a property and then enforces it at runtime
    - CFI, some aspects of bounds-checking, some types of randomizations, ...

- Others can use dynamic instrumentation
  - Stackguard, SFI (but may be limited if CFI can’t be assured)

- And yet others that cannot use static instrumentation
  - Obfuscated code, mainly malware
Obfuscation against Disassembly

- Conditional jumps where the condition is always true (or false)
  - Use an opaque predicate to hide this

- Instructions that fault
  - Execution continues in exception handler

- Embedding data in the midst of code
  - With indirect jumps that make it impossible to distinguish between code and data
Control-flow Obfuscation Against Reverse Engineering

- Split or aggregate
  - Basic blocks
  - Loops
    - e.g., one loop becomes two loops or vice-versa
  - Procedures
    - Replace one procedure by two or merge two procedures
    - Inline a procedure, or outline (i.e., create new procedure)

- Reorder

- Insert dead-code (i.e., unreachable code)
  - Obfuscate using conditions

- Replace instruction sequences w/ alternate ones
- Insert conditional jumps using “opaque” predicates
- Insert indirect jumps
- Exploit aliasing and memory errors
Data Obfuscation

- Rename variables
- Split or aggregate variables
  - Split structures into individual variables or vice-versa
- Split individual variables
  - E.g., $A = B - C$ – instead of $A$, use $B$ and $C$
  - Clone a variable
- Pad arrays (and possibly structures) with junk elements
- “Encrypt” data values
- Introduce extra levels of indirection
  - Instead of a simple variable, declare a pointer
- Introduce aliasing
- Introduce memory errors
- Introduce additional (or remove) function parameters
Covert Channels and Side-Channel Attacks
Covert Channels

- Confidential information may be leaked via channels that may be missed easily
  - Implicit flows in a program
  - Timing channels (network, cache, ...)
  - Steganographic techniques

- Examples
  - Transmit info by file name or metadata (e.g., timestamp)
    - Information retrieved by checking file presence or stat
      - No need to read the file (or have read permissions on the file)
  - “Port-knocking”
    - Transmit info by probing network ports in a certain sequence
  - Tcp acks or retransmissions, packet fragmentation, ...
Side-channel attacks

- Critical info may be leaked inadvertently
  - Error messages, e.g., invalid username vs password
  - Timing information
    - How long it took to verify a password, or encrypt something
    - Cache eviction attacks
    - Meltdown and Spectre attacks
  - Power-monitoring attacks
    - Use thermal imaging of a chip to monitor which circuits are being used and/or how much power is being used
    - Or simply monitor the power supply
  - Differential fault analysis
    - Force a particular fault (e.g., make a data line to be a “1” always) and examine how the program changes its behavior.
    - Rowhammer attacks on DRAM
  - Last two attacks motivate tamper-resistance in the context of building secure devices
    - Military equipment used in the field
    - Other devices that carry secrets and may be lost
Emanations

- **Electromagnetic emanations**
  - In old days, CRTs produced a lot of emanations that can be used to figure out what someone is doing from a distance

- **Keyboard emanations**
  - Researchers have shown it is possible to steal passwords using a microphone in a nearby office!

- **Power-line emanations**
  - Correlates fluctuations in power use (or EM waves on the powerline) with computations being performed

- **Snooping using telescopes**
  - Not just on-screen images, but reflections on a cup etc.
Remanence

- `malloc after free, or reuse of stack variables`
  - Exposes secrets that may be private to one program component to another.

- **Allocation of physical page for one process after it is used by another process**
  - Exposes secrets across processes
  - Can be avoided by immediately erasing confidential data
    - Beware: the compiler may eliminate this during optimization
    - Cache contents are flushed across process switch, so not a problem

- **Retained memory contents after power off**

- **Residual effects on hard drives**
  - may be data is just unlinked, not even overwritten
  - even after overwrite, it is often possible to recover old data
Intrusion Detection
Classes of Attacks

- **Probing: Reconnaissance before attack**
  - Port sweeps
  - OS/application finger printing

- **Denial of Service (DoS)**

- **Privilege escalation**
  - Remote to user
    - Attacker without any access to the victim machine gains access as a normal user, e.g., userid `nobody`
  - User to root
    - Attacker with access as normal user gains administrative privileges through an attack
  - These two privilege escalation attacks may be chained
  - Remote-to-user attacks typically exploit server applications (e.g., web server), while user-to-root attacks exploit other applications.
  - They are rarely caused by OS errors or errors in network protocol implementations
Intrusion Detection

- Some attacks will get through in spite of every protection measure. Intrusion detection is targeted to detect such attacks.
- Detection is a solution of last resort

- Assumption: Behavior of a system changes when it is subjected to attack
- Approach: Detect these changes in behavior
Intrusion Detection Issues

- **Detection rate**
  - What fraction of attacks are detected

- **False alarm rate**
  - May be measured in multiple ways
    - how many false alarms per day
    - what fraction of normal behavior is flagged as attack
    - what fraction of behavior reported as attack is not an attack (false alarm ratio)
  - Considerable disagreement on which measure to use
    - but the third criteria is probably the best
    - But IDS vendors (and may be researchers) don’t like it
      - Will you buy a system with FA rate of 98%?
      - But you may not mind 10 false alarms a day!
Intrusion Detection Techniques

- **Anomaly detection**
  - Use machine learning techniques to develop a profile of normal behavior
  - Detect deviations from this behavior
  - Can detect unknown attacks, but have high FA rate

- **Misuse detection**
  - Codify patterns of misuse
  - Attack behaviors usually captured using signatures
  - Can provide lower false alarm rate, but ineffective for unknown attacks

- **Behavior (or policy) based detection**
  - Specify allowable behavior, detect deviations from specifications
  - Can detect new attacks with low FA, but policy selection is hard
Intrusion Detection Algorithms

- **Pattern-matching**
  - Most commonly used in misuse and behavior based techniques

- **Machine-learning**
  - Statistical
  - Algorithmic
  - Neural networks and other techniques
Intrusion Detection Behaviors

Behaviors of

- Users
- Systems
  - processes, kernel modules, hosts, networks, …
Intrusion Detection Observation Points

- **Network-based (Network intrusion detection systems)**
  - **Benefits**
    - Unintrusive: plug a dedicated NIDS device on the network
    - Centralized monitoring
  - **Problems**
    - Encryption
    - Level of abstraction too low
    - Difference between data observed by NIDS and victim app.

- **Host-based**
  - Strengths/weaknesses complementary to NIDS
  - May be based on
    - system-call interception
    - audit logs and other log files
    - file system integrity (TripWire)
    - keystrokes, commands, etc.
Network Intrusion Detection

- Packet-based Vs Session-based
- Signature-based Vs Anomaly detection
- Example: SNORT (open source)
  - Uses pattern-matching on individual packets
- Some systems can block offending traffic
  - This is often dangerous, as systems usually have high false alarm rates
Host-based Intrusion detection

- System-call based characterizations most popular
- Behavior-based
  - System-call interposition plus wrappers
  - Domain/Type Enforcement
    - Certain application classes can access only certain files
    - Can prevent many privilege escalation attacks
    - Used in SELinux
- Anomaly detection
  - Sequences (finite-length strings) of system calls
  - FSA and PDA models of behavior
  - System call arguments
Automata Models for Learning Program Behaviors
Background

- Forrest et al showed that system call sequences provide an accurate and convenient way to capture security-relevant program behaviors
  - Subsequent research has further strengthened this result
- Key problem:
  - What is a good way to represent/learn information about system call sequences?
    - Issues: compactness, accuracy, performance, ...
Early Research

- Forrest et al [1999] compared several methods for learning system call sequences
  - Memorize subsequences of length N (N-grams)
  - Markov models
  - Data-mining (using RIPPER)

- N-grams found to be most appropriate
  - Markov models provided a slight increase in accuracy, but incurred much higher overheads
Illustration of N-gram Method

1.  S0;
2.  while (...) {
3.     S1;
4.     if (...) S2;
5.     else S3;
6.     if (S4) ... ;
7.     else S2;
8.     S5;
9.   }
10.  S3;
11.  S4;

Sample execution:
• S0  S1  S2  S4  S5
  S1  S3  S4  S2  S5  S3  S4
• S0  S3  S4

3-grams learnt:
• S0  S1  S2
• S1  S2  S4
• S2  S4  S5
• S4  S5  S1
• S5  S1  S3
• S1  S3  S4
• S3  S4  S2
• S4  S2  S5
• S2  S5  S3
• S5  S3  S4
Drawbacks of N-gram Method

- **Number of N-grams grows exponentially**
  - N must be small in practice (N=6 suggested)
  - Implication: difficult to capture long-term correlations
    - S₀ S₃ S₄ S₂ never produced by program, but all of the 3-grams in this sequence are

- **Remembers exact set of N-grams seen during training --- no generalization**
  - necessitates long training periods, or a high rate of false alarms
Models without Length Limitations

- **Finite-state automata**
  - Even an infinite number of sequences of unbounded length can be represented
  - Naturally capture program structures such as loops, if-then-else, etc.

- **Extended finite-state automata**
  - FSA + a finite number of state variables that can remember event arguments

- **Push-down automata**
  - By capturing call-return info:
    - PDAs are more accurate than FSM
    - Models are hierarchical and modular:
      - Hierarchical nature facilitates presentation
      - Smaller program models
      - Reuse of models for libraries
  - Extend PDAs to incorporate variables
Model extraction approaches

- **Static analysis [Wagner and Dean]**
  - **Pros:** conservative
  - **Cons:**
    - difficult to infer data values, e.g., file names
    - difficult to deal with libraries, dynamic linking, etc.
    - overly conservative
      - for intrusion detection, can detect only attacks that are outside of the semantic model used for analysis
      - specifically, buffer overflows, meta character attacks, etc.

- **Machine learning by runtime monitoring**
  - **Pros:**
    - can detect a much wider range of attacks
    - can deal with libraries, dynamic linking
    - inferring data values is easier
  - **Cons:**
    - False positives
Difficulty in Learning FSA from Strings

- Strings do not provide any information about internal states of an FSA
  - given S1  S2  S3  S2, which of the following FSA should we use?

```
S1 \rightarrow S2 \rightarrow S3 \rightarrow S2
```

```
S1 \rightarrow S2 \rightarrow S3
```

- what is the criteria for determining the “better” FSA?
- even if we can answer this, the answer will depend on additional examples
  - e.g., sequences S1 S2 and S1 S2 S3 S2 S3 S2 will suggest that the second FSA is the right one

- Learning FSA from sequences is computationally intractable
  [Kearns & Valiant 89, Pitt & Warmuth 89]
Key insight:

For learning program behaviors, additional information can be used to simplify the problem:

*exploit program counter value to obtain state information*
Learning FSA Models

A sample intercepted program behavior:

(S0,1) (S1,3) (S2,5) (S4,8) (S1,3) (S3,7) (S4,8) (S5,10)

1: s0;
2: while (...) {
3:   S1;
4:   if (...) 
5:     S2;
6:   else 
7:     S3;
8:   S4;
9: } 
10: S5;
Approach Details

- **Interception of system calls using ptrace (Linux)**
  - same mechanism used by Forrest and other researchers

- **Examine process stack to obtain program counter information**

- **Dynamic linking poses a problem**
  - same function may be loaded at different locations during different runs
  - Solution: use program counter value corresponding to the code calling the dynamically loaded library
  - Side benefit: ignoring library behavior makes FSA more compact
Approach Details (Continued)

- **Fork**: Parent and child monitored with same FSA, but process contexts maintained.
- **Exec**: typically, a new FSA for the execve’d program is used.
- **Detection time**
  - mismatch may occur in terms of either the system call or program location
  - use leaky bucket algorithm for aggregation
  - program counter helps resynchronize even after observing behavior not seen during training
Training Convergence

- FSA method converges faster than N-grams
  - roughly speaking, FSA method can do with roughly an order of magnitude less training period than N-gram method
False Positive Rate

- FP results are similar to convergence
  - for a given FP rate, FSA method requires an order of magnitude less training than N-gram method
Mimicry Attacks

- Attacks crafted with knowledge of IDS
- Execute only system call sequences that would be permitted by the model
- A mimicry attack can be developed from an attack sequence by inserting “junk” system calls that make it appear as if a legitimate sequence is generated
  - Junk system calls made possible by using bad system call arguments
- Graybox IDS complicate mimicry attacks due to the need to fake call site
  - Control does not return to attack code after a call!
- But can still be made to work
- Known mimicry attacks based on memory corruption+injected code
Learning System Call Arguments

- Earlier methods focus on control-flows
  - System call sequences (N-grams)
  - Automata models of behavior
    - FSA or PDA, with transitions labeled with system calls
  - System call arguments largely ignored

- Detects usual control-hijack attacks

- Don’t detect most attacks that modify resources access by a system call
  - Non-control data attacks
  - Race condition attacks
  - Mimicry attacks
  - …
Approach

- Incorporate dataflow info into control-flow models
  - Exploit control-flow context to improve model precision
  - Go beyond treating system call arguments in isolation
    - “A program can only write a compressed version of its input file”
      - open(“/home/joe/model.ps”, “r”), …, open(“/home/joe/model.ps.gz”, “w”), …
    - “All sensitive files should be closed before execve operation”
      - open(“/etc/passwd”, “r”) = 6, …, close(6), …, execve(“untrusted_prog”, …), …
  - Learning temporal relations => parameterized models
    - command ”find –exec /bin/ls…” resulting in execve(“/bin/ls”, …)

- Learning-based:
  - learns dataflows that seem to be present in program
Need for Control-flow Context

- The dataflow properties need control-flow context
  - Example:
    - L1: `fd1 = open("/etc/passwd", O_RDONLY);`
    - L2: `fd2 = open("/tmp/out", O_RDWR);`
    - We cannot combine information about opened filenames, otherwise "/etc/passwd" at L2 will be accepted

- Control-flow context provided by previous FSA/PDA techniques could be used
- Context encoded by giving names for event arguments
Motivating Example: \textit{simpletar} program

```c
int main(int argc, char **argv)
{
    source_dir = argv[1], target_file = argv[2];
    opendir("/opt/proj")
opendir("/opt/proj/src")

    start("/opt/proj", "/tmp/proj.tar")
    open("/tmp/proj.tar", WR) = 3

    isdirectory("/opt/proj/README")
    open("/opt/proj/README", RD)=4
    read(4, …)
    write(4, …)
    close(4)

    while ((dir_name = pop()) != NULL) {
        isdirectory(dir_name)
        opendir(dir_name)

        foreach (dir_entry)
        {
            if (isdirectory(dir_entry))
                push(dir_entry);
            else {
                source_fd = open(dir_entry, RD);
                read(source_fd, buf);
                write(target_fd, buf);
                close(source_fd);
            }
        }
    }

    close(target_fd);
}
```

```
open("/tmp/proj.tar", WR) = 3
```
Possible Dataflow Relationships

- **Unary Relations:** properties of a single system call argument
  - Represented as $X \mathcal{R} c$, where
    - $X$: an argument name,
    - $c$: a constant value,
    - $\mathcal{R}$: a unary relation
  - Examples of unary relation $\mathcal{R}$:
    - **equal** => $X$ takes only a single value always equal to $c$
    - **elementOf** => $X$ takes any value from the set $c$
    - **subsetOf** => $X$ takes values all of which are drawn from set $c$
    - **range** => $X$ takes values in the range $c$ (e.g., $c = (0, 2)$)
    - **isWithinDir** => $X$ is a file name argument that is always contained within a specified directory $c$
      - Example: If $X$ takes values “/home/user/abc” and “/home/user/xyz”, we can say that $X$ **isWithinDir** “/home/user”
**Possible Dataflow Relationships**

- **Binary relations**: captures relationships between two event arguments
  - Represented as $X \mathcal{R} Y$, where
    - $X, Y$: argument names
    - $\mathcal{R}$: a binary relation
  - Examples of binary relation $\mathcal{R}$:
    - **equal** => equality between $X$ and $Y$
    - **isWithinDir** => file/directory $X$ is within directory $Y$
    - **contains** => directory $X$ contains file/directory $Y$
    - **hasSameDirAs** => $X$ and $Y$ are within a common directory
    - **hasSameBaseAs** => $X$ and $Y$ have same base (eg: a.c, a.h)
    - **hasSameExtensionAs** => $X$ and $Y$ have same extension (eg: a.c, b.c)
Binary Relations

How to interpret a binary relation?

- A naïve interpretation of \( X \text{ equal } Y \): \( X \) and \( Y \) have only one possible value in all traces
  - not useful interpretation when \( X \) and \( Y \) occur multiple times

Our approach: \( X \ R \ Y \Rightarrow X \) is related to closest preceding \( Y \)

- \( X \ R_T Y \): For every occurrence of \( X \) in a trace \( T \), \( X \ R Y \) holds

  Example: For a labeled trace \( T \) of the form:
  \[
  Y = 1, \ Z = 2, \ X = 1, \ Y = 2, \ X = 2,
  \]
  we say that \( X \text{ equal}_T Y \), but not \( Y \text{ equal}_T X \)

- \( X \ R_T Y : \forall T \in \mathcal{T}, \ X \ R_T Y \) holds
Learning Unary Relations

- With each argument, maintain a list of all values encountered in all traces
- If too many values are associated with an argument, use relations that approximates set
  - range (min-max pair) for port numbers, file descriptors,
  - isWithinDir for file names
  - subsetOf for flag arguments of open
- Runtime and storage requirements
  - A trace of size $N$ (in bytes) can be learnt in $O(N)$
  - Storage requirement: $O(S)$, where $S$ is size of control flow behavior model
Learning Binary Relations

- **Key algorithmic issue:** How to find pairs of arguments of relationship in near-linear time?
  - Trace lengths (N) can be $10^5$ to $10^7$
  - Quadratic algorithms too expensive

- **Approach:**
  - Equality relationships
    - Store most recent values of arguments in a hash table
    - Given the current value of an argument, simply look up in the hash table for the set of all candidate arguments
  - String relationships
    - Use trie data structure to achieve similar effect

- **Algorithm complexity is O(Nr)** where r is maximum number of relations involving any single argument
  - Much smaller than $N^2$ – typically $r < 10$
Effectiveness for Attack Detection

- Can detect data corruption attacks that don’t involve control-flow hijack
- Race condition attacks
- Allow verifying nontrivial security props
  - If the IDS doesn’t raise alarms then these properties will be preserved
- Examples of properties verified:
  - `find` executes only those programs that are specified using a “-exec” command-line option
  - All files read by `tar` would reside within the directory specified on the command-line
  - The only file written by `gzip` is obtained by adding “.gz” suffix to its argument
Effectiveness for Attack Detection

- Fingerd symlink vulnerability
  - Attack exploits absence of symbolic link check
    ```c
    if (lstat(tbuf, &sbuf1)) return 0;
    L1: fd = open(tbuf, O_RDONLY); ...
    fp = fopen(fd, "r"); ...
    ```
  - Attacker makes `.plan` as a symbolic link to `/etc/passwd`
  - **Attack detection**: violation of binary relationship between username command line argument and file opened at L1
## False Alarm Rates

<table>
<thead>
<tr>
<th>Program</th>
<th>Training Trace Length (#syscalls)</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trace Length (#syscalls)</td>
<td>Base (X 10^{-5})</td>
</tr>
<tr>
<td>httpd</td>
<td>(X 10^{6}) 1.75 3.10</td>
<td>16.60</td>
</tr>
<tr>
<td>sshd</td>
<td>4.75 14.74</td>
<td>0.35</td>
</tr>
</tbody>
</table>

- Unary relations increase false alarms modestly
- Binary relations add high false alarms in httpd
  - 95% of these are due to accidental relationships learnt for a rarely occurring system call in training phase.
  - We can address them by adding a notion of confidence level with each relation
Models are very small as compared to size of programs

- Program sizes: 68KB(find) to 435KB(wu-ftpdp)
  - #Binary relations are linearly proportional to size of control flow model
Summary of Argument-Learning Approach

- Uses dataflow information to enhance precision of intrusion detection models
- Can be layered over other techniques for learning control flows
- Effective in detecting sophisticated attacks
- Extracts models that are compact and produce low false alarms
- Enables formal reasoning about security guarantees of models
Defending Software Systems from Cyber Attack Campaigns

R. Sekar
Secure Systems Lab
Stony Brook University

Joint work with: Sandeep Bhatkar, Lorenzo Cavallaro, Dan DuVarney, Ajay Gupta, Niranjan Hasabnis, Md Nahid Hossain, Zhenkai Liang, Sadegh Momeni, Huan Nguyen, Riccardo Pelizzi, Soumyakant Priyadarshan, C.R. Ramakrishnan, Qiao Rui, Prateek Saxena, Scott Stoller, Weiqing Sun, Wai-Kit Sze, Laszlo Szekeres, Alok Tongaonkar, Tung Tran, Prem Uppuluri, V.N. Venkatakrishnan, Wei Xu, Yves Younan, and Mingwei Zhang

Research funded in part by: AFOSR, DARPA, ONR and NSF.
Cyber Attacks Continue to Escalate ...
A Timeline of Advanced Attack Campaigns

1. **Titan Rain**
   - Targeting US military (called the greatest transfer of wealth in history) stealing blueprints of planes, space-based lasers, missile navigation and nuclear submarines

2. **Stuxnet**
   - Targeting Nuclear Facilities (called first Digital Weapon)

3. **Target**
   - 40 million payment card credentials and 70 million customer records lost

4. **Yahoo**
   - Information associated with at least 500 million user accounts was stolen

5. **OPM**
   - Described by federal officials as among the largest breaches of government data in the history of the United States

6. **Deep Panda**
   - Targeting Health Care Services (breach of financial and medical records of up to 80 million customers)

7. **Equifax**
   - Exposed the names, SSN, birth dates, addresses, and, in some instances, driver’s license numbers of about 44 percent of the current American population

8. **Marriott**
   - Hundreds of millions of customer records, including credit card and passport numbers, being exfiltrated by the attackers
**Observations:**

- Initial compromise relies on **vulnerability exploitation** and/or **social engineering**
- Most steps require attackers to **deploy** and **execute** custom malware
Our Goal: Layered Defense Targeting Each Phase

- **Identify (and fix) vulnerabilities before deployment**
  - fuzzing, symbolic execution, model-based analysis, ...

- **Prevent exploitation of vulnerabilities that remain**
  - Memory corruption, e.g., buffer overflows
  - Input validation, e.g., SQL and command injection, cross-site scripting (XSS), ...

- **Restrict malware behavior to limit damage**
  - policy-based confinement of untrusted code

- **Quickly detect attack campaigns that evade all defenses**
  - Real-time attack campaign reconstruction from log data
Memory Corruption: King of all vulnerabilities

Example shows a stack smashing exploit. Memory corruption can target any memory area (stack, heap, or static memory).

- **Popular**, as it allows attackers to inject and execute code of their choice.
- **Widely prevalent** due to large size of low-level code (C/C++/assembly) on today’s systems.
- **Difficult to eliminate**, as processors (CPUs) operate at low level (machine code, pointers, interrupts, ...)

```c
void f(const int *A, int n) {
    int buf[100];
    int i = 0;
    while (i < n) {
        buf[i] = A[i++];
    }
    ...
}
```
Diversity: A Generic Defense Against all Memory Corruptions

- Software bugs are difficult to detect or fix
  - **Question**: Can we make them harder to exploit?

**Benign Diversity**

- Preserve functional behavior
  - On benign inputs, diversified program behaves exactly like the original program
- Randomize attack behavior
  - On inputs that exercise a bug, diversified program behaves differently from the original
C/C++ languages don’t specify locations of objects (code or data) in memory.

But attackers need to know them.
Address Obfuscation (2003)

- C/C++ languages don’t specify locations of objects (code or data) in memory.
  - But attackers need to know them.
- Address obfuscation makes object locations hard to predict
  1. Randomize **base address** of stack, heap, code, and static area
  2. Random gaps between stack frames
  3. Fine-grained randomization of heap allocations
Address Obfuscation (2003)

- C/C++ languages don’t specify locations of objects (code or data) in memory.
  - But attackers need to know them.

- Address obfuscation makes object locations hard to predict
  1. Randomize base address of stack, heap, code, and static area
  2. Random gaps between stack frames
  3. Fine-grained randomization of heap allocations

Technique #1, invented by us and several others, is widely deployed now (Linux, Windows, Mac, Android). Ours is the first paper on this idea.
Fine-grained Address-Space Randomization (2005)

- (Coarse-grained) ASLR is susceptible to information-leaks
  - Base address can be revealed by a single pointer value
- Fine-grained randomization mitigates leaks by randomizing *relative order of objects*
(Coarse-grained) ASLR is susceptible to information-leaks
- Base address can be revealed by a single pointer value

Fine-grained randomization mitigates leaks by randomizing *relative order of objects*
- Permutes order of functions, static variables and local arrays at load time
  - implement using an extra level of indirection!
- Combined with a separate *safe stack* for non-arrays.
  - *Now available in the LLVM compiler*

Fine-grained randomization is the topic of numerous recent efforts
Data-Space Randomization

- **C/C++ languages don’t specify** *bit-level representation* of data.
  - But attackers need to know them.

- **Data space randomization makes low-level representations hard to predict**
  - Assign a random bit mask for each object, xor with default representation
    - When there is overflow from one object to the next, the second object gets corrupted unpredictably
  - Modest overheads (20%)
  - Needs to assign same masks for possibly aliased objects
Memory Corruption Exploits: The complete story

Make pointer out-of-bounds
- Make pointer dangling
  - Use pointer to write
  - Use pointer to read

- Modify a data pointer...
  - ... to attacker specified value
    - Dereference corrupted ptr.

- Modify code ...
  - ... to attacker specified code

- Modify a code pointer...
  - ... to target code address
    - Use pointer by indir. call/jmp
      - Exec. gadgets or functions
    - Use pointer by ret instruction
      - Execute injected code

- Modify data ...
  - ... to attacker specified value
    - Use corrupted data variable

- Output data
  - Interpret the leaked value

Code corruption

Control-flow hijack

Data-only attack

Information leak
Coarse-grained Address-Space Randomization

Address-space Randomization [USENIX 2003]

- Make pointer out-of-bounds
- Make pointer dangling

Make pointer dangling
- Use pointer to write
- Use pointer to read

Use pointer to read
- Modify a data pointer
- Modify code
- Modify a code pointer

Modify a code pointer
- Specify value
- Specify code
- Code address

Specify code
- Dereference corrupted ptr.
- Use pointer by indir. call/jmp
- Use corrupted data variable
- Exec. gadgets or functions
- Execute injected code

Use corrupted data variable
- Data-only attack
- Information leak

Data-only attack
- Interpret the leaked value

Interpret the leaked value
- Output data
- Modify data...

Modify data...
- ...to attacker specified value

Execute injected code
- Control-flow hijack
- Code corruption
Data-Space Randomization

- Make pointer out-of-bounds
- Make pointer dangling

- Use pointer to write
- Use pointer to read

- Modify a data pointer
  - ... to attacker specified value
  - Dereference corrupted ptr.

- Modify code
  - ... to attacker specified code

- Modify a code pointer
  - ... to attacker specified code

- Modify data
  - ... to attacker specified code

- Output data
  - Interpreting the leaked value

Data-space Randomization [DIMVA 2008]

- Use pointer by indir. call/jmp
- Use pointer by ret instruction
- Use corrupted data variable
- Exec. gadgets or functions
- Execute injected code

- Code corruption
- Control-flow hijack
- Data-only attack
- Information leak
Bounds-Checking

Maintain metadata and use it to validate array bounds and pointer values
Code Pointer Integrity

Apply memory safety selectively to protect code pointers
Control Flow Integrity

Prevent control-flow hijack (transfer to attacker-intended code)
Input Validation Bugs
A model for input validation bugs

### Incoming Request
(Untrusted input)

- $send_to_list = \$_GET['sendto']$
- $command = "gpg -r$send_to_list 2>&1"
- `popen($command)`

### Program

- `sendto="nobody; rm -rf *"`
- `$command="gpg -r nobody; rm -rf * 2>&1"
- `popen($command)`

### Outgoing Request/Response
(Security-sensitive operations)
(To backend servers, command interpreters, DBs, files...)

**Attack:** Removes files
A model for input validation bugs

Detect exertion of control:

Use **taint**

- Tainted text shown in *red*, trusted text in black.

**Incoming Request**
(Untrusted input)

$send_to_list = \$_GET['sendto']$

sendto="nobody; rm -rf *"

**Program**

$command = "gpg -r$
$send_to_list 2>&1"

$command="gpg -r
nobody; rm -rf * 2>&1"

popen($command)

**Outgoing Request/Response**
(Security-sensitive operations)
(To backend servers, command interpreters, DBs, files...)

Detect if control is intended:

Use **policies**

To be practical, policies should be application independent.
A model for input validation bugs

Detect exertion of control:
Use **taint**
- Tainted text shown in *red*, trusted text in black.

Detect if control is intended:
Use **policies**
- To be practical, policies should be application independent
Effectiveness of Taint-Enhanced Policy Enforcement

Our Approach defeats most exploits
Common Weakness Enumeration (CWE)/SANS Top 25 Software Errors list ranks these vulnerabilities at #1 through #4, #7, and #9

Our approach protects from exploits of all these vulnerabilities
Our prototype blocked every one of a dozen exploits on popular web apps.

Compiled from MITRE’s Common Vulnerability Enumeration (CVE) for 2010
http://cve.mitre.org

Also at NIST’s National Vulnerability Database
http://nvd.nist.gov/
Source-code or binary instrumentation:
- We developed some of the most efficient techniques in this regard
  - 50% on C-source (USENIX Sec 2006)
  - 100% on binaries (CGO 2008)

Taint-inference: (NDSS 2009)
- Match input and output using *approximate substring matching*
- **Black-box**: No need to modify source or binaries
- **Efficient**: Often, less than 5%
Policy for Command and Script Injections

- Attacks cause *structural changes* due to *tainted data*
- **Lexical confinement**: Tainted data can’t span multiple tokens
Policy for Command and Script Injections

- **Attacks cause** *structural changes* due to *tainted data*
- **Lexical confinement:** Tainted data can’t span multiple tokens
- **Subtree confinement:** Tainted data can’t *overflow* into adjacent subtree
- One policy applicable across command languages (SQL, JavaScript, Perl, Shell, ...)

**Taint- and Syntax-aware policies**

1. **SpanNodes policy:** captures "lexical confinement"
   - Tainted data to be contained within a single leaf node
2. **StraddleTrees policy:** captures "overflows"
   - Tainted data begins in the middle of one subtree, flows into next
Cross-site Scripting Protection for Browsers

**XSSFilt (2012):** Was deployed on *Pale Moon* in 2016, a Firefox clone with 500K users.

**SrvFilt (2017):** Browser-independent, *evasion-resistant*, blocks *Dom-XSS*. 
Binary Instrumentation
Why Binaries?

- Unavailability of source code
- Ease of deployment
- **Completeness:** Low-level libraries and hand-written assembly
- **Soundness:** Compiler optimizations can eliminate security-critical code
Challenges of Working With Binaries

- **Size and complexity of instruction sets such as x86 and ARM.**
  - Techniques often limited to a single processor
  - Only a subset of instructions supported

- **High performance overheads**
  - Dynamic instrumentation (e.g., Pin) is robust, but slow.
  - *Static instrumentation can be fast, but faces challenges on large/complex binaries.*
Overcoming Challenges: Instruction Set Complexity

- Modern instruction sets are complex
  - Intel’s manual is 1500+ pages and 1100+ instructions
  - ARM’s manual is over 1000 pages (and growing!)
  - *Frequent additions of ISA extensions*

- Manual modeling is tedious, error-prone, and impossible to keep up with
  - Most existing tools support only the top one or two architectures.
  - What about non-main-stream processors, e.g., in IoT environments?
Overcoming Challenges: Instruction Set Complexity

- Modern compilers (e.g., GCC) can generate code for numerous architectures
  
  1. *Source* → *IR*: Translate source code to architecture-neutral intermediate representation
  
  2. *IR* → *Asm*: Translate IR to assembly using architecture-specific *machine descriptions*

- IR contains detailed semantics that has been is extensively tested

- Question: Can we reverse the IR to assembly translation process?
  
  - Lifts assembly to a common IR that is simpler to analyze
LISC: Learning Instr. Semantics from Compilers [ASPLOS ’16]

- Black-box approach: does not depend on gcc internals
- Learns Asm $\rightarrow$ RTL (gcc’s IR) mapping from examples
  - Almost an endless supply of examples available!
  - LISC learns a decision tree with variables
    - Not a standard classification problem: we are learning a function
    - Must ensure sound translation in all cases
LISC Approach

1. Collect training data
   - Compile many packages to collect \( \langle \text{rtl}, \text{asm} \rangle \) pairs
LISC Approach

1. Collect training data
   - Compile many packages to collect \( \langle \text{rtl}, \text{asm} \rangle \) pairs

2. Parameterize: for each pair \( \langle \text{rtl}, \text{asm} \rangle \)
   - Parse \text{rtl} and \text{asm} into trees
   - identify the parameters (leaves)
   - compute the mapping between them

\[
\langle \text{sub } 34, \%rbx \rangle
\]

\[
\langle \text{set } (\text{reg } \text{rbx}) \text{ (plus } (\text{reg } \text{rbx}) \text{ (const } -34)) \rangle
\]
LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs

2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse $rtl$ and $asm$ into trees
   - Identify the parameters (leaves)
   - Compute the mapping between them

$\langle \text{sub } 34, \% \text{rbx} \rangle$

$(\text{set (reg } \text{rbx} \rangle \langle \text{plus (reg } \text{rbx} \rangle \langle \text{const - 34 } \rangle))$
LISC Approach

1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs

2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse $rtl$ and $asm$ into trees
   - Identify the parameters (leaves)
   - Compute the mapping between them

\[ \langle \text{sub } 34, \% \text{rbx} \rangle (\text{set}(\text{reg } \text{rbx}), (\text{plus}(\text{reg } \text{rbx}), (\text{const} - 34))) \]
LISC Approach

1. **Collect training data**
   - Compile many packages to collect \(\langle \text{rtl}, \text{asm} \rangle\) pairs

2. **Parameterize: for each pair \(\langle \text{rtl}, \text{asm} \rangle\)**
   - Parse \(\text{rtl}\) and \(\text{asm}\) into trees
   - Identify the parameters (leaves)
   - Compute the mapping between them

3. **Build transducer from parameterized pairs**
   - Transducer is an automaton similar to Moore/Mealy machine
   - Input is \(\text{asm}\) tree, output is \(\text{rtl}\) tree
Transducer Construction Example

\[
\text{add } \%\text{ebx}, \%\text{eax} \rightarrow (\text{set (reg eax)} (\text{plus (reg eax)} (\text{reg ebx})))
\]
\[
\text{add } 5, \%\text{eax} \rightarrow (\text{set (reg eax)} (\text{plus (reg eax)} (\text{const } 5)))
\]
\[
\text{sub } 2, \%\text{eax} \rightarrow (\text{set (reg eax)} (\text{plus (reg eax)} (\text{const } -2)))
\]

\[
(\text{set (reg X)} (\text{plus (reg X)} (_)))
\]

\[
\begin{align*}
\text{add} & \quad \text{sub} \\
X = \%\text{eax} & \quad X = \%\text{eax}, Y = 2 \\
& \quad (\text{const } -1*Y)
\end{align*}
\]
\[
\begin{align*}
\text{Y} = 5 & \quad \text{Y} = \%\text{ebx} \\
& \quad (\text{const Y}) \\
& \quad (\text{reg Y})
\end{align*}
\]
LISC: Evaluation

- **Completeness:**
  - 99.5% of x86 and 99.8% of ARM instructions achieved
    - after training with about 10 chosen binaries
  - Remaining are mostly NOPs and other obsolete instructions (e.g., BCD arithmetic)

- **Soundness:**
  - Proved under reasonable assumptions
    - context-independent translation of RTLs into assembly
  - Also experimentally verified on core instructions

- **Now LISC v2 supports x86_64**
  - Work done originally on x86_32
Static Binary Instrumentation: Challenges

- Robust static disassembly
  - Including low-level libraries and hand-written assembly

- Static instrumentation without breaking complex code
  - Position-independent code, C++ exceptions, Signal handlers, ...

- Secure instrumentation
  - Ensure instrumentation of all code
  - Ensure that added security checks cannot be bypassed
Static Binary Instrumentation: BinCFI Solution

- Robust static disassembly
  - Error-detecting and error-correcting disassembly
  - Errors detected by following (direct and indirect) control-flow targets

- Static instrumentation without breaking complex code
  - *Key problem:* Instrumentation changes code locations, so find and adjust all code pointers.
  - We show that runtime code pointer translation is both robust and efficient

- Secure instrumentation
  - based on *control-flow integrity*
    - What You Disassemble Is What You eXecute
BinCFI Results [USENIX Sec ’13] (Best paper award)

- **Supports large and low-level COTS (“stripped”) binaries**
  - glibc, Firefox, Adobe Reader, gimp, etc.
    - Over 300MB of (intel 32-bit) binaries in total.

- **Eliminates 99% of control-flow targets and 93% of possible gadgets**
  - Remaining gadgets provide very limited capability

- **Good performance while providing full transparency**
  - About 10% overhead on CPU-intensive C-benchmarks, somewhat higher for C++ programs
Most of BinCFI’s overhead comes from runtime code pointer translation

**Question:** Can we avoid this runtime translation?

- Requires code pointers to be translated at instrumentation time
Most of BinCFI’s overhead comes from runtime code pointer translation

**Question:** Can we avoid this runtime translation?

- Requires code pointers to be translated at instrumentation time

**Yes:** For 64-bit position-independent binaries

- Almost all code on modern Linux distributions falls in this category
- Pointers are all explicitly identified in these binaries
  - but there is no information on which of these point to code
Most of BinCFI’s overhead comes from runtime code pointer translation

**Question:** Can we avoid this runtime translation?

- Requires code pointers to be translated at instrumentation time

**Yes:** For 64-bit position-independent binaries

- Almost all code on modern Linux distributions falls in this category
- Pointers are all explicitly identified in these binaries
  - but there is no information on which of these point to code

**Approach:** Develop static analysis to distinguish code and data pointers

- Relies on detailed instruction semantics derived using our LISC work
x86_64 Instrumentation: Evaluation summary

- Languages: C, C++, Fortran, Assembly
- Compilers: gcc/g++, llvm (clang/clang++), gfortran
- Functionality:
  - Reboot OS after instrumenting core libraries (e.g., glibc and libpthread)
  - Instrument and test ~ 200 shared libraries and applications, cumulative size: 197MB
    - For comparison: size all binaries on a default install of Ubuntu is 940MB
  - Instrument and test SPEC CPU 2017 benchmarks, cumulative size: 941MB
- Performance: Zero overhead
Fine-grained Code Randomization for x86_64 (Ongoing Work)

**Approach**
- Develop techniques for breaking up and reordering binary code blocks
- Implement using our address-translation free static instrumentation

**Results**
- *Low overheads:* 0% for the cheapest technique to 3.5% for the most secure technique
- Scales to large and complex binaries — tested on over 200MB of binary code, much of it in low-level libraries.
Attack Scenario Reconstruction
APT Campaigns: Overview and Challenges

APT Campaigns combine social engineering with advanced exploits to penetrate high-value networks, stay hidden for months (e.g., Equifax breach).

**Stitch Together:**
- Phishing
- Exploit vulnerability
- Malicious web
- Exploit browser
- RAT
- Network scan
- Malware propagation
- Code Repo
- Database
APTs combine social engineering with advanced exploits to penetrate high-value networks, stay hidden for months (e.g., Equifax breach)

- “Needle-in-a-haystack:” billions of events per day, but often, just one in a million is an attack
- “Connecting the dots:” Stitch together isolated steps to present a graphical summary (“storyline”) of the entire campaign in real-time
- DARPA carried out 5 Red Team evaluations over 2016–19.
Illustrative Example

- **Attacker goal:** Insert backdoor into a vendor’s software

- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
**Illustrative Example**

- **Attacker goal:** Insert backdoor into a vendor’s software

- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob’s `.bashrc` to redefine `sudo`
Illustrative Example

- **Attacker goal:** Insert backdoor into a vendor’s software

- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob’s `.bashrc` to redefine `sudo`
  3. Next time Bob uses `sudo`, it copies `/home/bob/crt1.o` to `/lib/crt1.o`
Illustrative Example

- **Attacker goal:** Insert backdoor into a vendor’s software

- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob’s `.bashrc` to redefine `sudo`
  3. Next time Bob uses `sudo`, it copies `/home/bob/crt1.o` to `/lib/crt1.o`
  4. When Alice builds her software, malicious `crt1.o` code is included in her executable.
**Illustrative Example**

- **Attacker goal:** Insert backdoor into a vendor’s software

- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob’s `.bashrc` to redefine `sudo`
  3. Next time Bob uses `sudo`, it copies `/home/bob/crt1.o` to `/lib/crt1.o`
  4. When Alice builds her software, malicious `crt1.o` code is included in her executable.
  5. When this software is run, it exfiltrates sensitive data (password file)
Our goal is to automate all steps, all the way to the rendered scenario graph.
Provenance Tags

Trustworthiness (t-tag)

**Benign:** Data from sources believed to be benign.

**Unknown:** No good basis to trust this source.

Confidentiality (c-tag)

**Secret:** Highly sensitive, e.g., `/etc/shadow`

**Private:** Loss may not pose a direct security threat.

**Public:** Widely available, e.g., on public web sites

Code Vs Data Trustworthiness

- Processes have two t-tags: *code t-tag* and *data t-tag*
- Separation (a) aids detection and (b) speeds analysis by focusing on fewer root causes
**Untrusted exec (UE):** Subject w/ high code trustworthiness execs lower t-tag object.

**Suspicious modification (SM):** Subject with lower code tag modifies higher t-tag file.

**Data leak (DL):** Untrusted subject writes confidential data to network.

**Untrusted execution preparation (UP):** Memory/file objects with low data trustworthiness made executable.
Attack Detection Policies

Untrusted exec (UE): Subject w/ high code trustworthiness execs lower t-tag object.

Suspicious modification (SM): Subject with lower code tag modifies higher t-tag file.

Data leak (DL): Untrusted subject writes confidential data to network.

Untrusted execution preparation (UP):
Memory/file objects with low data trustworthiness made executable.
**Attack Detection Policies**

**Untrusted exec (UE):** Subject w/ high code trustworthiness execs lower t-tag object.

**Suspicious modification (SM):** Subject with lower code tag modifies higher t-tag file.

**Data leak (DL):** Untrusted subject writes confidential data to network.

**Untrusted execution preparation (UP):** Memory/file objects with low data trustworthiness made executable.
Attack Detection Policies

Untrusted exec (UE): Subject w/ high code trustworthiness execs lower t-tag object.

Suspicious modification (SM): Subject with lower code tag modifies higher t-tag file.

Data leak (DL): Untrusted subject writes confidential data to network.

Untrusted execution preparation (UP): Memory/file objects with low data trustworthiness made executable.
**Untrusted exec (UE):** Subject w/ high code trustworthiness execs lower t-tag object.

**Suspicious modification (SM):** Subject with lower code tag modifies higher t-tag file.

**Data leak (DL):** Untrusted subject writes confidential data to network.

**Untrusted execution preparation (UP):** Memory/file objects with low data trustworthiness made executable.
**Untrusted exec (UE):** Subject w/ high code trustworthiness execs lower t-tag object.

**Suspicious modification (SM):** Subject with lower code tag modifies higher t-tag file.

**Data leak (DL):** Untrusted subject writes confidential data to network.

**Untrusted execution preparation (UP):** Memory/file objects with low data trustworthiness made executable.
Goal: Identify entry point of an attack.

- Entry point is a source, i.e., vertex with in-degree zero.

Starting points: Suspect vertices marked by attack detectors.

Problem: Find source vertices from which a suspect vertex is reachable.

Complications: Multiple sources, and multiple suspect nodes
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
  - and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
- and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm

`129.55.33.44:80` `firefox` `/home/bob/.bashrc` `cp` `bash` `sudo` `sudo` `/lib/crt1.o` `ld` `apt-get` `/home/alice/test` `/home/alice/test.o` `/home/alice/test.c` /home/alice/crt1.o

EXEC

UE: /home/alice/test
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
- and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
- and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
- and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
  - and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
- and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Backward Analysis: Key Ideas

- Prefer shorter paths over longer ones
- Favor paths that avoid redundant edges
- Prefer edges corresponding to flow of untrusted code
  - and, to a lesser extent, untrusted data
- Preference encoded using a custom edge-weight function to Dijkstra’s shortest path algorithm
Forward Analysis

**Goal:** Identify attack impact, in terms of all objects/subjects affected by the attack.

- Generate a subgraph of provenance graph that only includes objects and subjects affected by the attack.

**Starting point:** Sources identified by backward analysis

**Challenge:** Straight-forward dependence analysis may yield a graph with hundreds of thousands (if not millions) of edges.
Forward Analysis: Key Ideas

- Use cost metric to prune off distant nodes, i.e., nodes at a distance $\geq d_{th}$
Forward Analysis: Key Ideas

- Use cost metric to prune off distant nodes, i.e., nodes at a distance $\geq d_{th}$
- Cost metric favors edges with untrusted code trustworthiness (cost=0);
- and, to a lesser extent, edges with untrusted data trustworthiness (cost=1)
Forward Analysis: Key Ideas

- Use cost metric to prune off distant nodes, i.e., nodes at a distance $\geq d_{th}$
- Cost metric favors
  - edges with untrusted code trustworthiness (cost=0);
  - and, to a lesser extent, edges with untrusted data trustworthiness (cost=1)
In-memory exploit of firefox

Relies on preexisting malware on the system (a kernel device and an application)
Graph Generated for a Ransomware Campaign
Summary of Campaign Reconstruction Results

- **New tag and policy-based detection** targets essential attack goals, e.g., steal proprietary data
  - Detected 174/175 attack steps in DARPA’s Red Team Eng #1 [USENIX Sec 2017]

- **Fast and effective forensic algorithms**
  - Analyzed days’ worth of data in minutes, filtered out over 99.999% of events

- **Compact in-memory storage** using *novel dependency reduction algorithms* [USENIX Sec 2018]
  - GBs of logs use 10’s of MBs of memory

- **Can map attack campaigns to attacker’s goals** (foothold establishment, lateral movement, exfiltration, etc.) [S&P 2019]

- Produces compact scenario graphs even for **stealthy campaigns that use zero-days, rootkits, and preexisting malware** [S&P 2020]
Summary: Layered Defense Against Cyber Attack Campaigns

- Identify (and fix) vulnerabilities before deployment
  - fuzzing, symbolic execution, model-based analysis, ...
- Prevent exploitation of vulnerabilities that remain
  - Memory corruption defenses
    - randomization
    - memory safety, ...
  - Input validation, e.g., SQL and command injection, cross-site scripting (XSS), ...
  - Binary instrumentation
- Restrict malware behavior to limit damage
  - policy-based confinement of untrusted code
- Quickly detect attack campaigns that evade all defenses
  - Real-time attack campaign reconstruction from log data