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 ...

ZDNet VIDEOS WINDOWS 10 ENTERPRISE SOFTWARE CLOUD AI SECURITY TR PREMIUM MORE

MUST READ: The UK's tech unicorns are booming, but the next wave won't find it so easy

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.

BBC Sign in News Sport Reel Workli

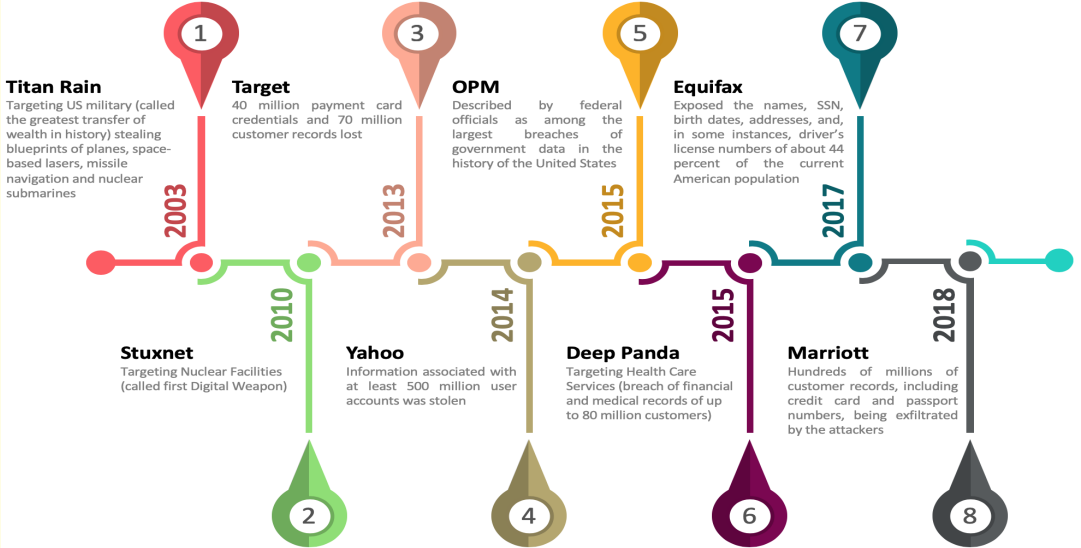
NEWS

Home Video World UK Business Tech Science Stories

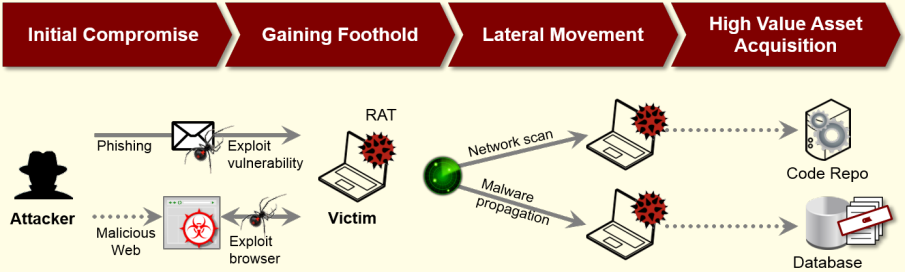
Technology

WhatsApp discovers 'targeted' surveillance attack

A Timeline of Advanced Attack Campaigns



Phases of a Typical Attack Campaign



Observations:

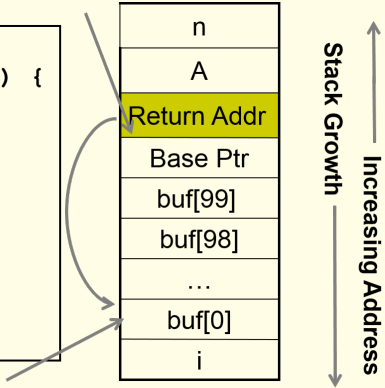
- Initial compromise relies on vulnerability exploitation and/or social engineering
- Most steps require attackers to deploy and execute custom malware

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

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



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, ...)

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

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 (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*

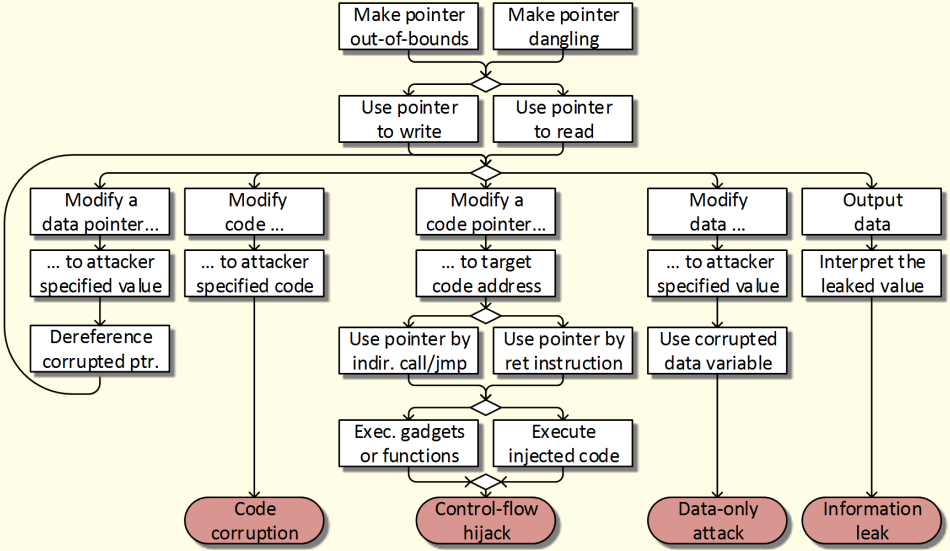
Fine-grained Address-Space Randomization (2005)

- (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*
- 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

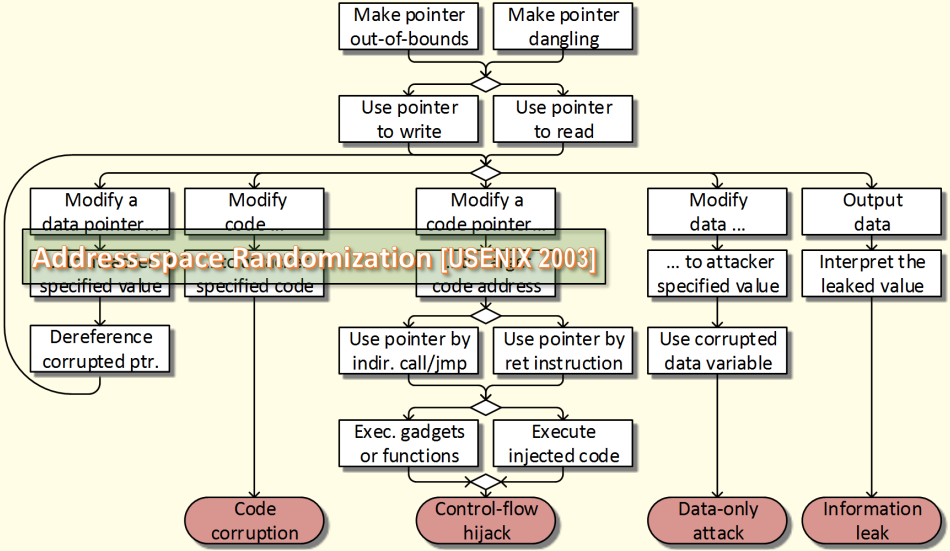
Data-Space Randomization

- 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

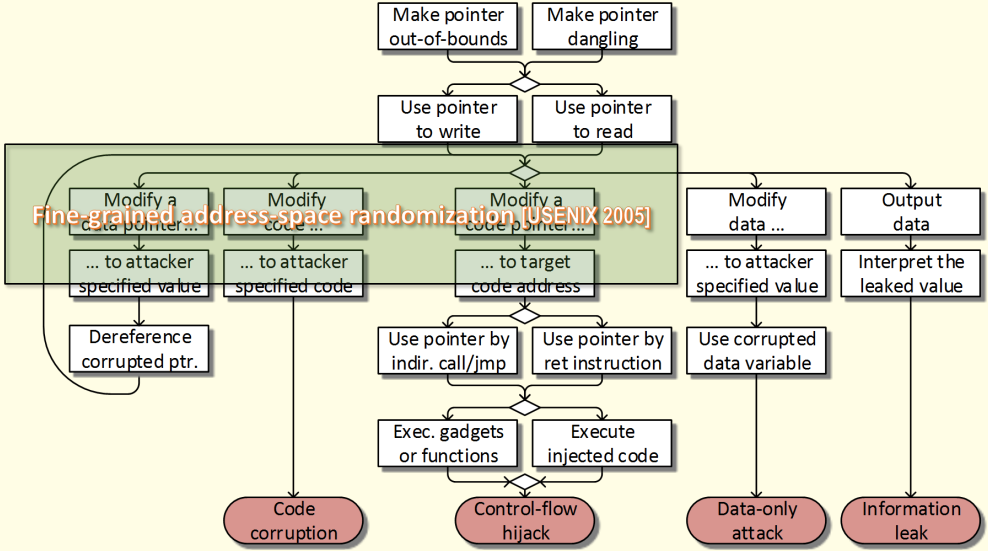


Coarse-grained Address-Space Randomization



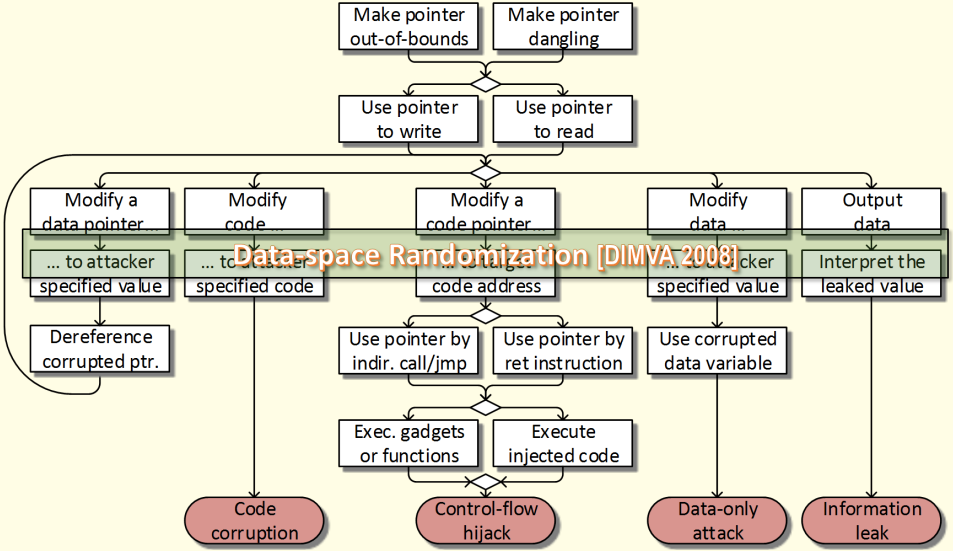
Address-space Randomization [USENIX 2003]

Fine-grained Address-Space Randomization



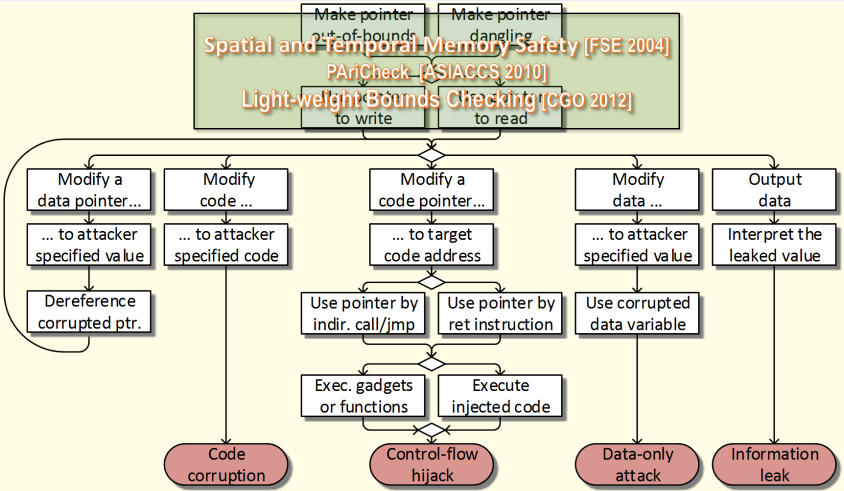
Fine-grained address-space randomization [USENIX 2005]

Data-Space Randomization



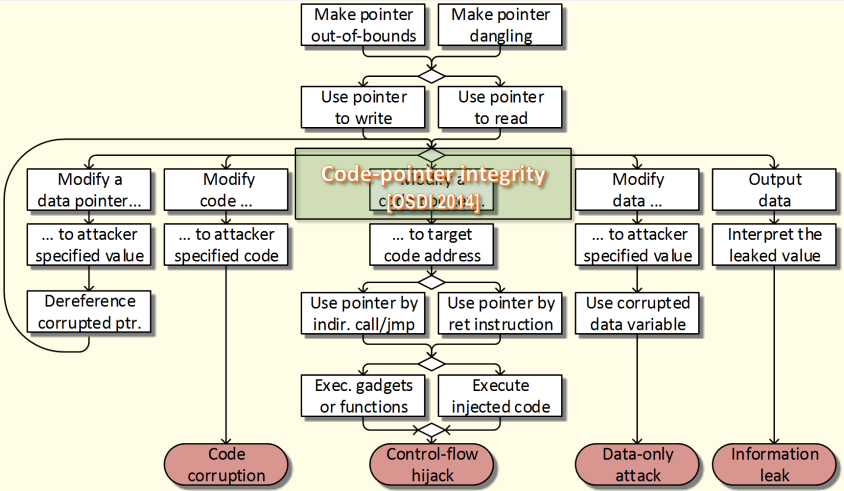
Data-space Randomization [DIMVA 2008]

Bounds-Checking



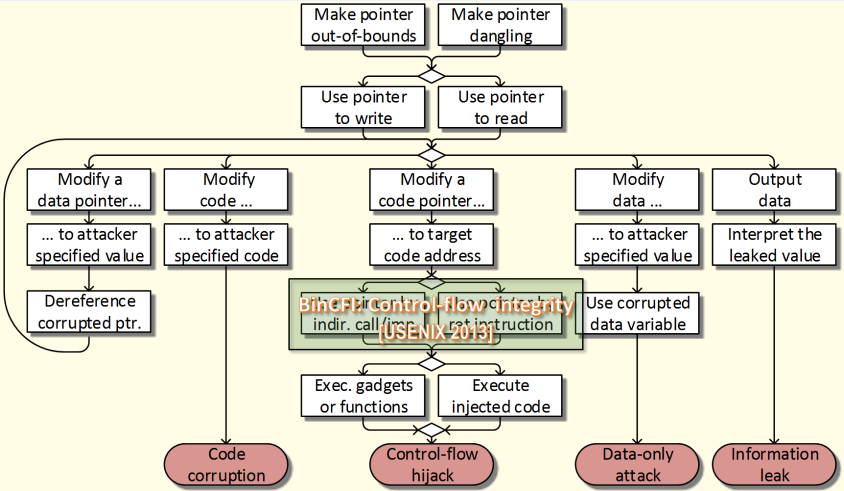
Maintain metadata and use it to validate array bounds and pointer values

Code Pointer Integrity



Apply memory safety selectively to protect code pointers

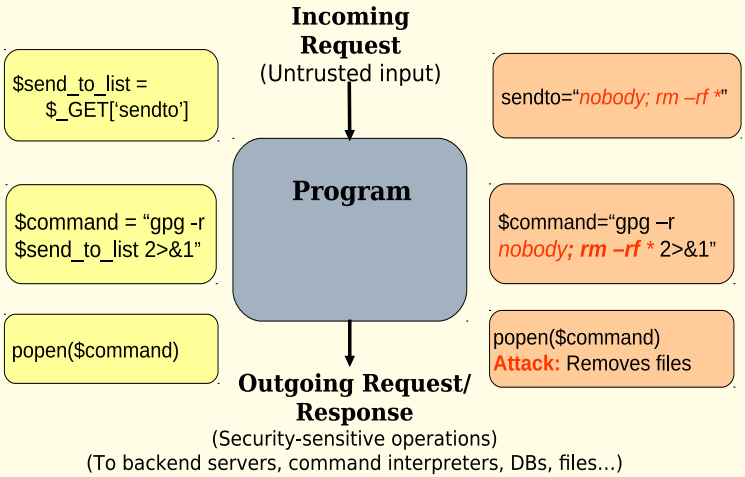
Control Flow Integrity



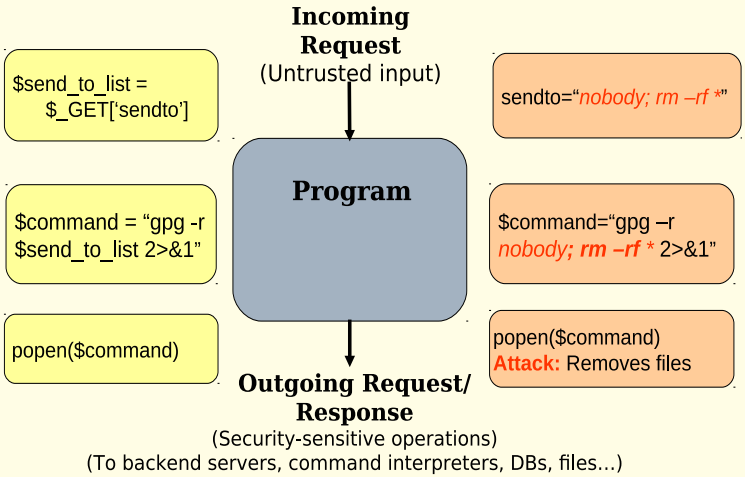
Prevent control-flow hijack (transfer to attacker-intended code)

Input Validation Bugs

A model for input validation bugs



A model for input validation bugs

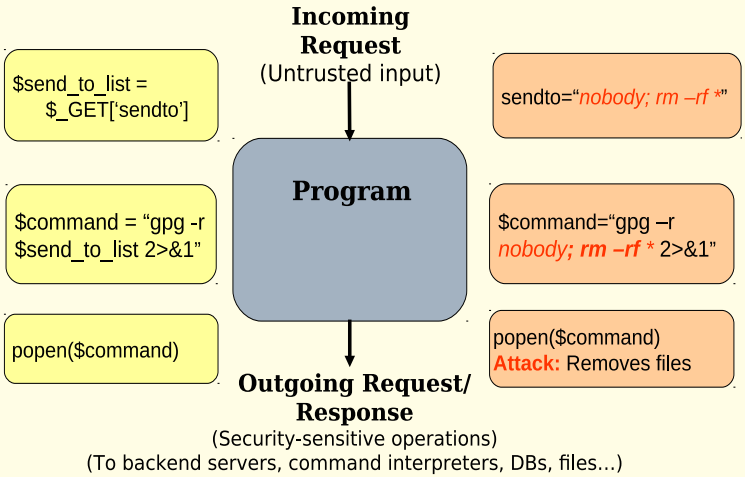


Detect exertion of control:

Use **taint**

- Tainted text shown in *red*, trusted text in black.

A model for input validation bugs



Detect exertion of control:

Use **taint**

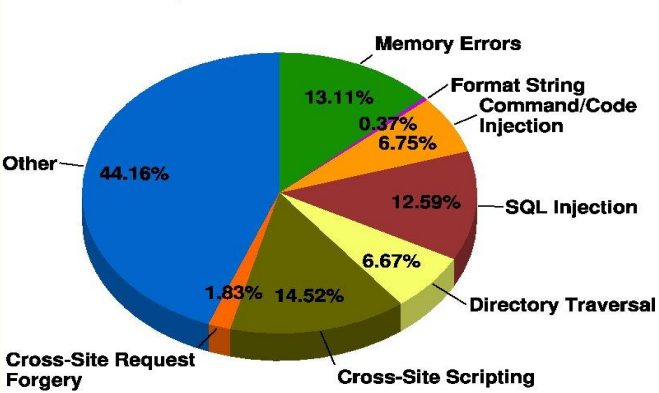
- Tainted text shown in *red*, trusted text in black.

Detect if control is intended:

Use **policies**

- To be practical, policies should be application independent

Effectiveness of Taint-Enhanced Policy Enforcement



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/

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.

Efficient Taint Computation

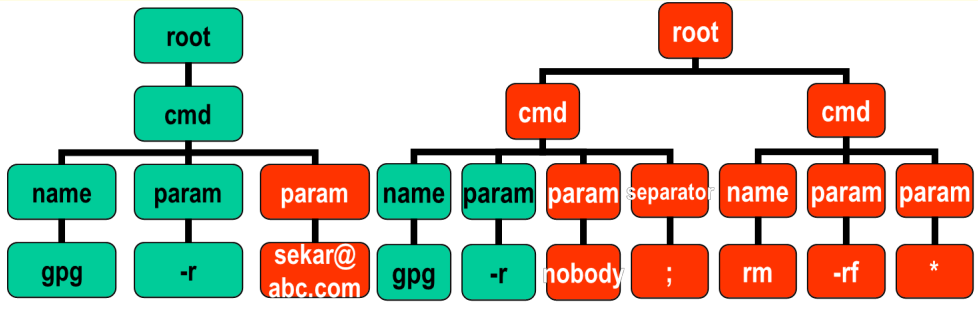
Source-code or binary instrumentation:

- 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)

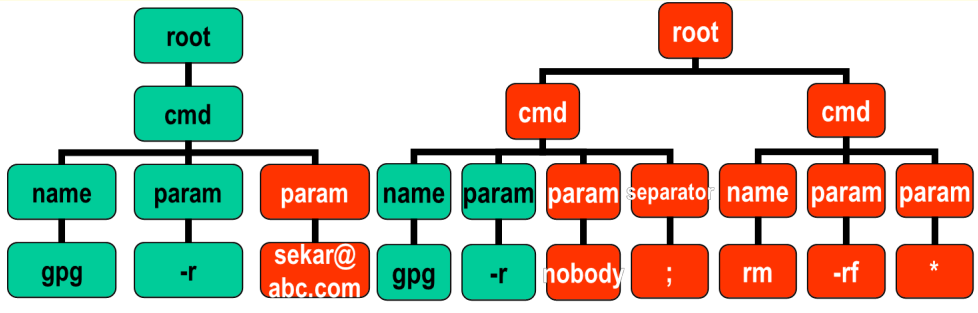
- Match input and output using *approximate substring matching*
- *Black-box:* No need to modify source or binaries
- *Efficient:* Often, less than 5%

Policy for Command and Script Injections



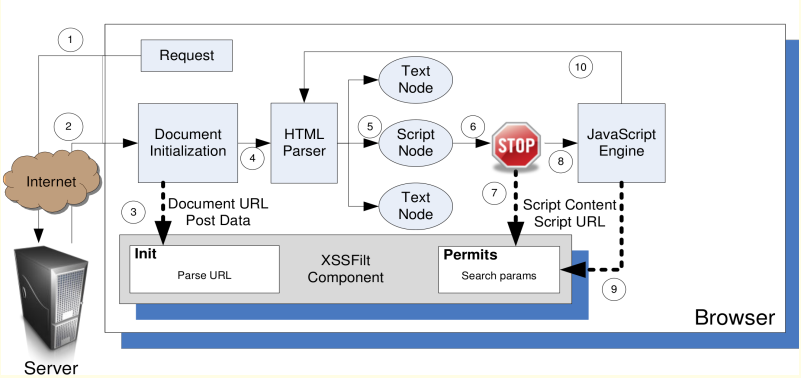
- Attacks cause *structural changes* due to *tainted data*
- **Lexical confinement:** Tainted data can't span multiple tokens

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, ...)

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* $\longrightarrow$ *IR:* Translate source code to architecture-neutral intermediate representation
  2. *IR* $\longrightarrow$ *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 → 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 $\langle rtl, asm \rangle$ pairs

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

⟨sub $34, %rbx
(set (reg rbx) (plus (reg rbx) (const -34)))⟩

LISC Approach

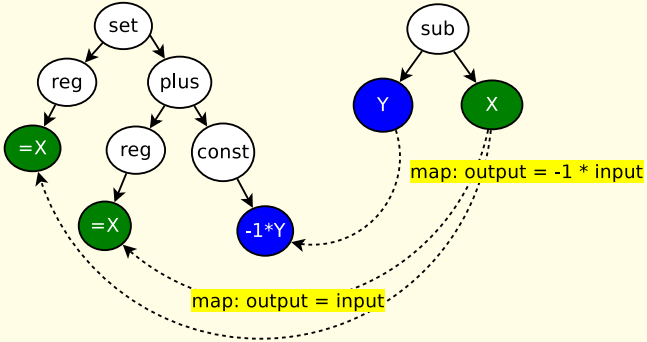
1. Collect training data
   - Compile many packages to collect $\langle rtl, asm \rangle$ pairs
2. Parameterize: for each pair $\langle rtl, asm \rangle$
   - Parse *rtl* and *asm* into trees
   - identify the parameters (leaves)
   - compute the mapping between them

$\langle$sub $ 34 , % rbx
(set (reg rbx) (plus (reg rbx) (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

⟨sub $ 34 , % rbx
(set (reg rbx) (plus (reg rbx) (const - 34)))⟩

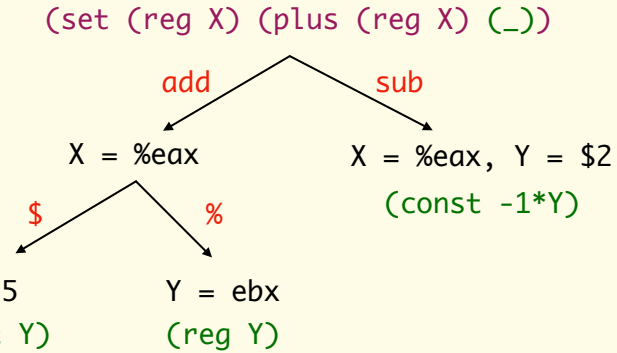


LISC Approach

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
3. Build transducer from parameterized pairs
   - transducer is an automaton similar to Moore/Mealy machine
   - input is *asm* tree, output is *rtl* tree

Transducer Construction Example

```
add %ebx, %eax → (set (reg eax) (plus (reg eax) (reg ebx)))
add $5, %eax   → (set (reg eax) (plus (reg eax) (const 5)))
sub $2, %eax   → (set (reg eax) (plus (reg eax) (const -2)))
```



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
  - Download from `http://seclab.cs.sunysb.edu/seclab/liscV2/`

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

Static Instrumentation: Further Performance Improvements...

- 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

Static Instrumentation: Further Performance Improvements...

- 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

Static Instrumentation: Further Performance Improvements...

- 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

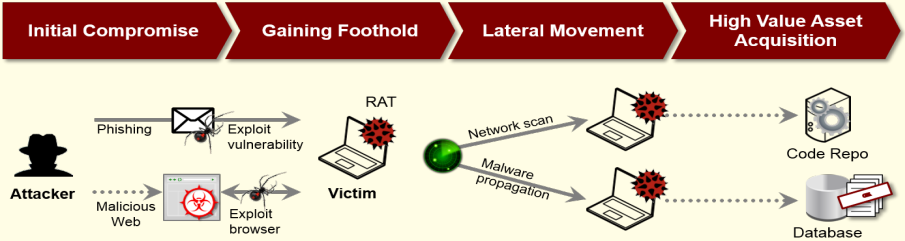
- Languages: C, C++, Fortran, Assembly
- Compilers: gcc/g++, llvm (clang/clang++), gfortran
- Functionality:
  - Reboot OS after instrumenting core libraries (e.g., `glibc` and `libpthread`)
  - Instrument and test $\sim$ 200 shared libraries and applications, cumulative size: 197MB
    - For comparison: 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.

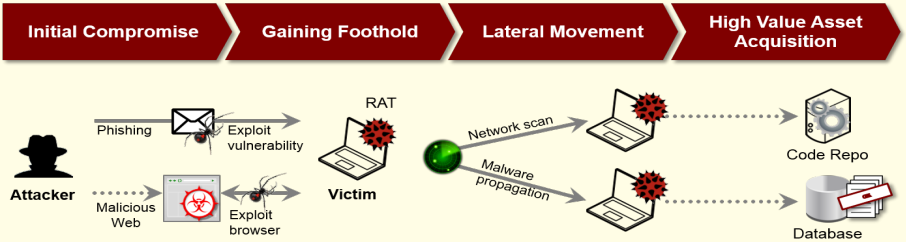
Attack Scenario Reconstruction

APT Campaigns: Overview and Challenges



APTs combine social engineering with advanced exploits to penetrate high-value networks, stay hidden for months (e.g., Equifax breach)

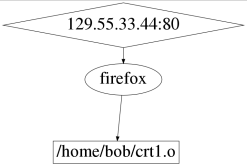
APT Campaigns: Overview and Challenges



APTs combine social engineering with advanced exploits to penetrate high-value networks, stay hidden for months (e.g., Equifax breach)

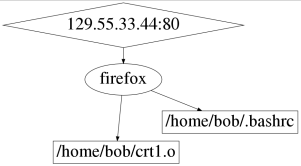
- "Needle-in-a-haystack:" *billions of events* per day, but often, just *one in a million* is an attack
- "Connecting the dots:" Stitch together isolated steps to present a *graphical summary* ("storyline") of the entire campaign in *real-time*
- DARPA carried out 5 Red Team evaluations over 2016–19.

Illustrative Example



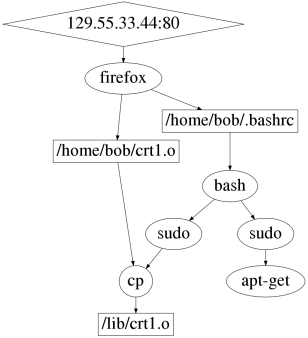
- **Attacker goal:** Insert backdoor into a vendor's software
- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`

Illustrative Example



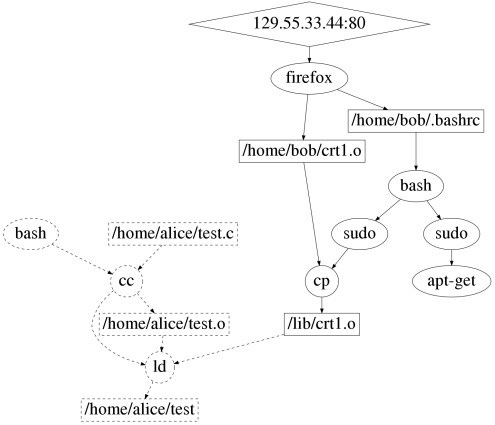
- **Attacker goal:** Insert backdoor into a vendor's software
- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob's `.bashrc` to redefine `sudo`

Illustrative Example



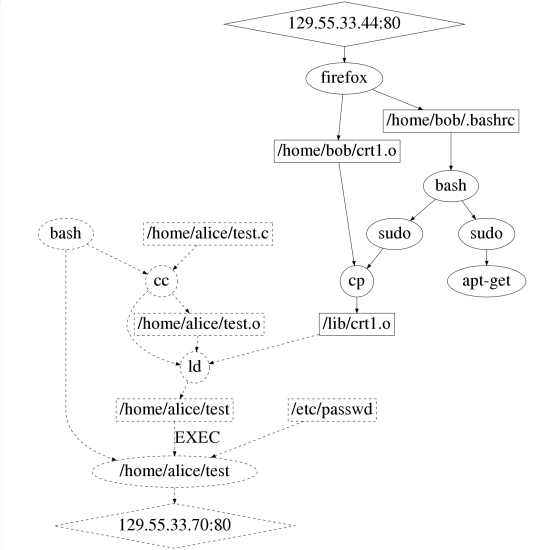
- **Attacker goal:** Insert backdoor into a vendor's software
- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob's `.bashrc` to redefine `sudo`
  3. Next time Bob uses sudo, it copies `/home/bob/crt1.o` to `/lib/crt1.o`

Illustrative Example



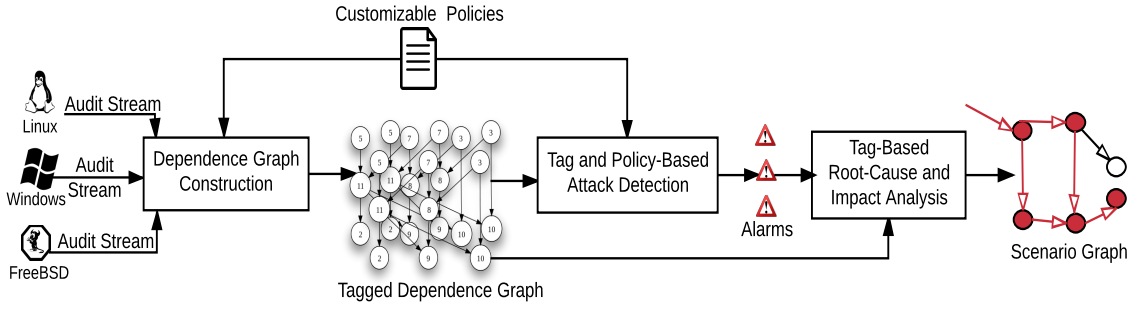
- **Attacker goal:** Insert backdoor into a vendor's software
- **Steps:**
  1. Use a browser vulnerability to drop a malicious version of `crt1.o` in `/home/bob`
  2. Modify Bob's `.bashrc` to redefine `sudo`
  3. Next time Bob uses sudo, it copies `/home/bob/crt1.o` to `/lib/crt1.o`
  4. When Alice 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)

System Architecture



- 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.

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

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

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.

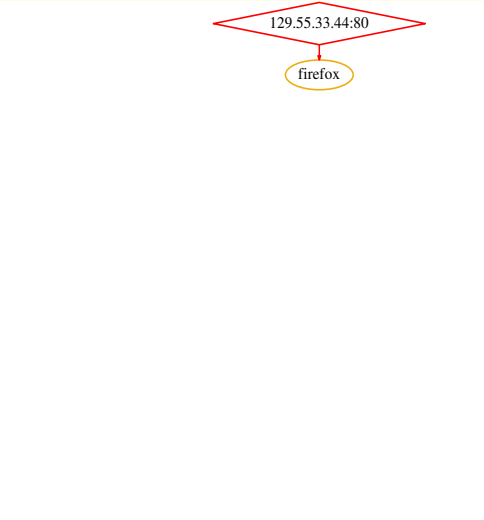
Attack Detection Policies

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.



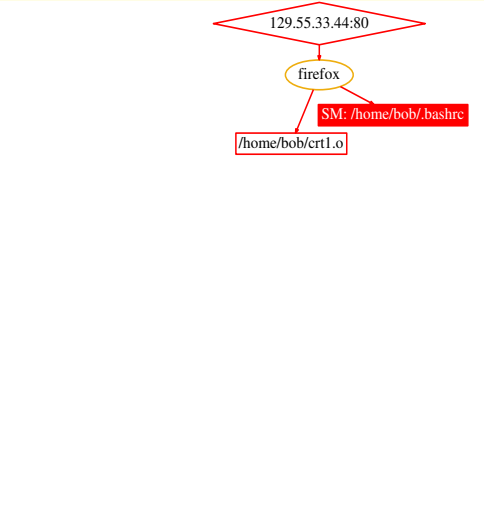
Attack Detection Policies

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.



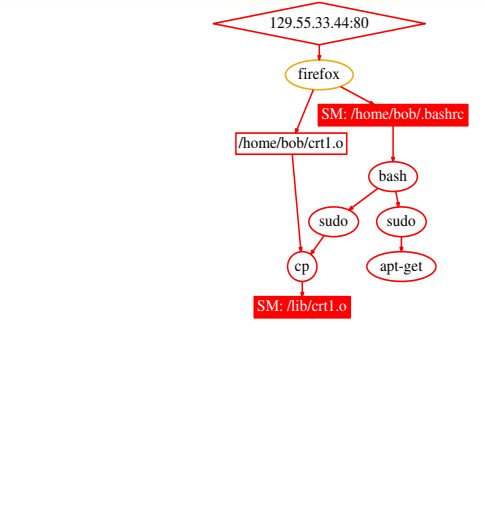
Attack Detection Policies

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.



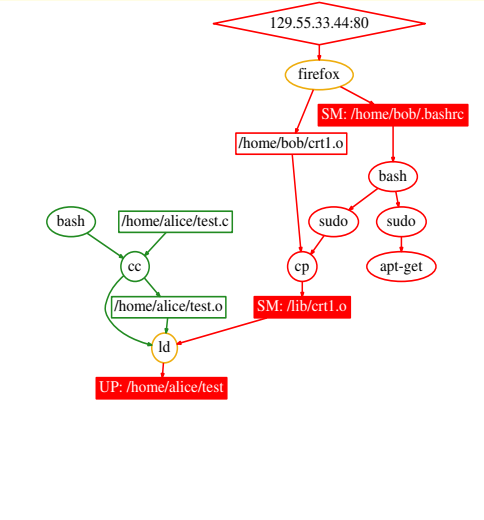
Attack Detection Policies

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.



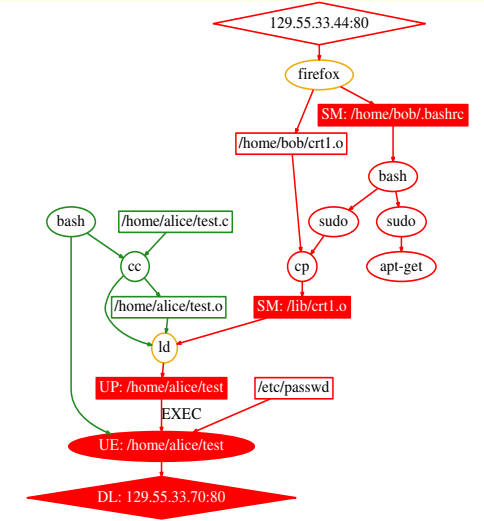
Attack Detection Policies

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.



Backward Analysis

Goal: Identify entry point of an attack.

- Entry point is a *source*, i.e., vertex with in-degree zero.

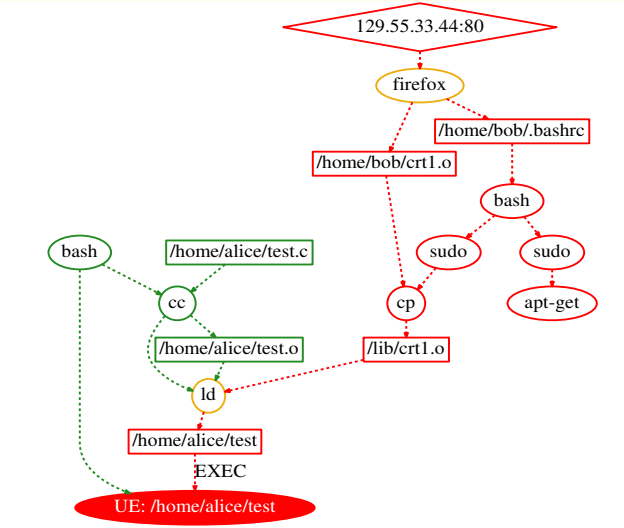
Starting points: Suspect vertices marked by attack detectors.

Problem: Find source vertices from which a suspect vertex is reachable.

Complications: Multiple sources, and multiple suspect nodes

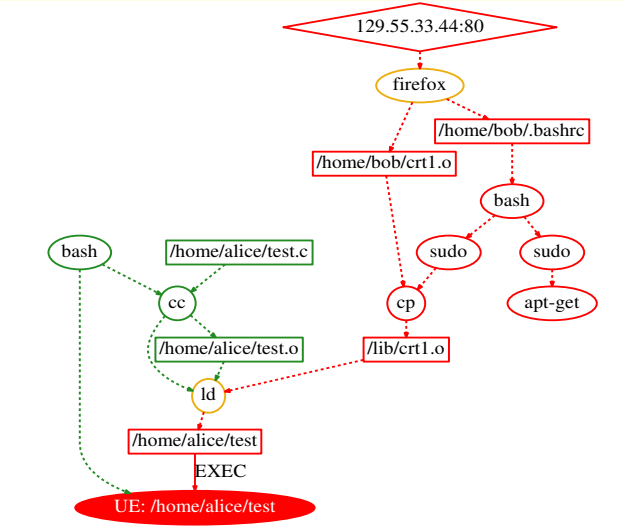
Backward Analysis: Key Ideas

- 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



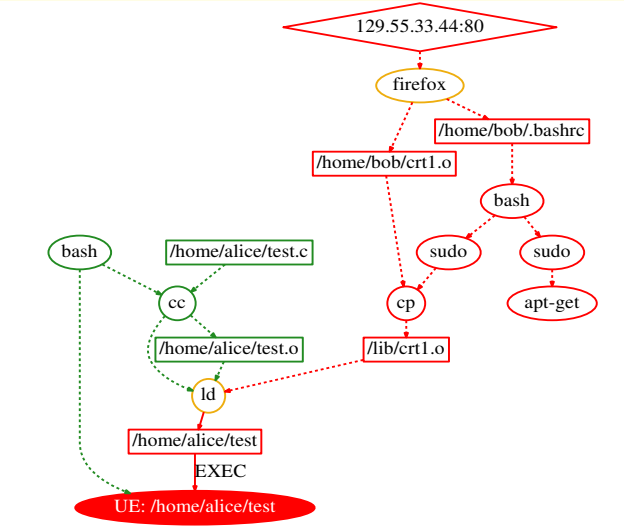
Backward Analysis: Key Ideas

- 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



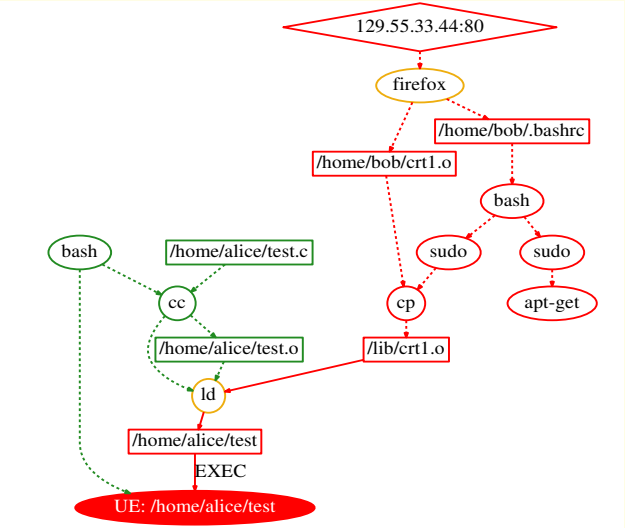
Backward Analysis: Key Ideas

- 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



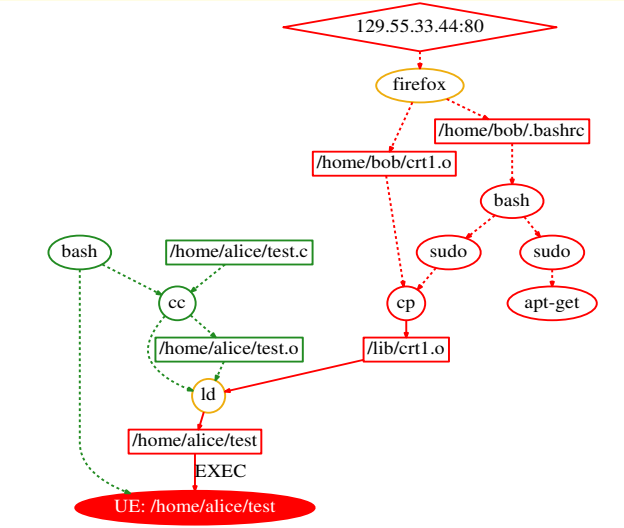
Backward Analysis: Key Ideas

- 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



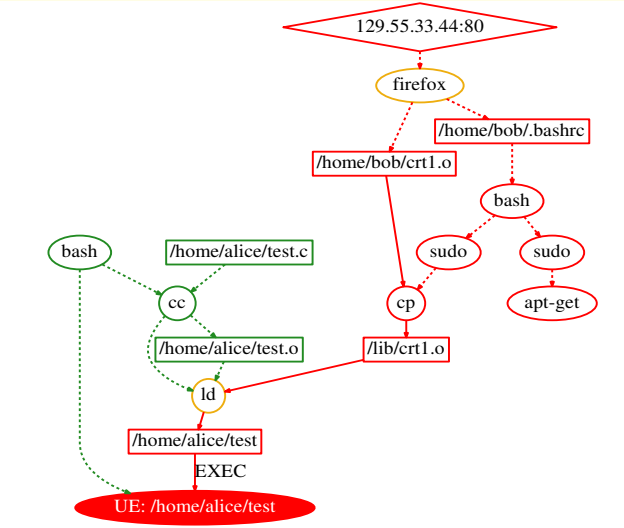
Backward Analysis: Key Ideas

- 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



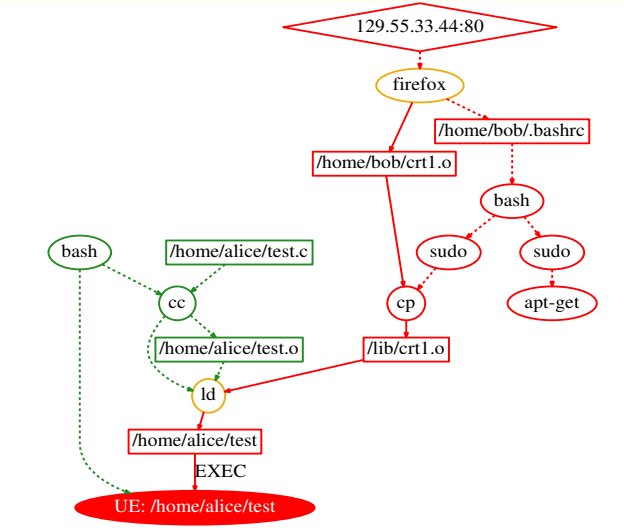
Backward Analysis: Key Ideas

- 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



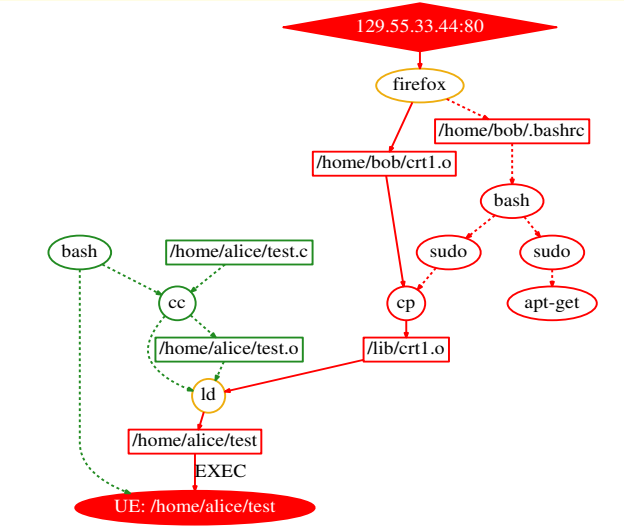
Backward Analysis: Key Ideas

- 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.

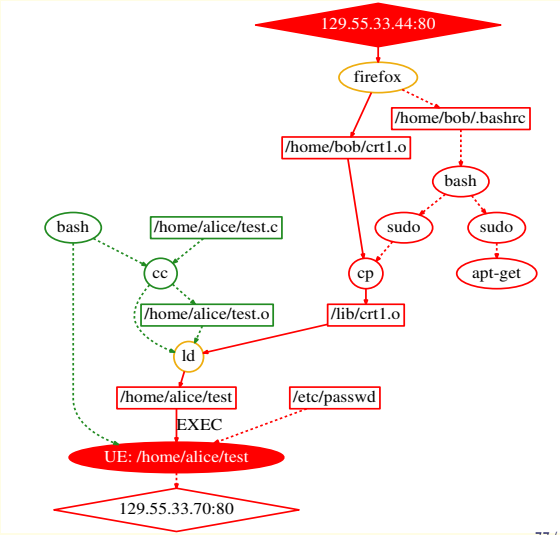
- Generate a subgraph of provenance graph that only includes objects and subjects affected by the attack.

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.

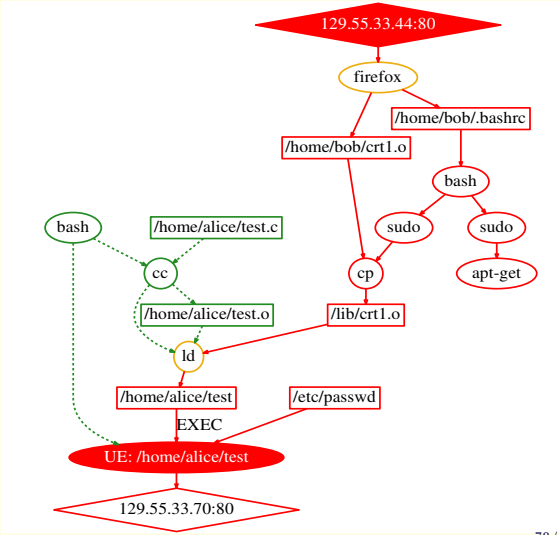
Forward Analysis: Key Ideas

- Use cost metric to prune off distant nodes, i.e., nodes at a distance $\geq d_{th}$



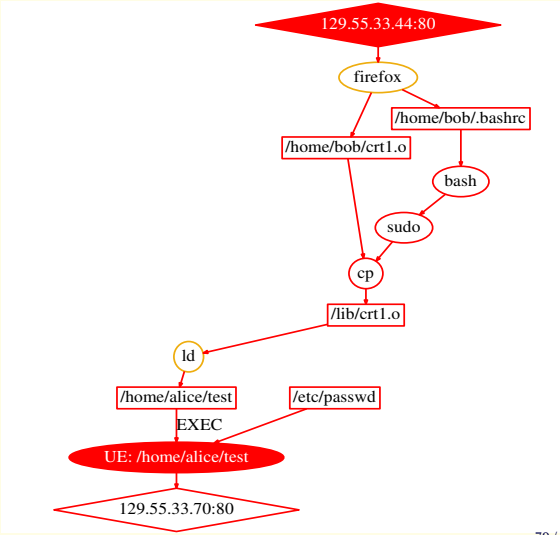
Forward Analysis: Key Ideas

- 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)

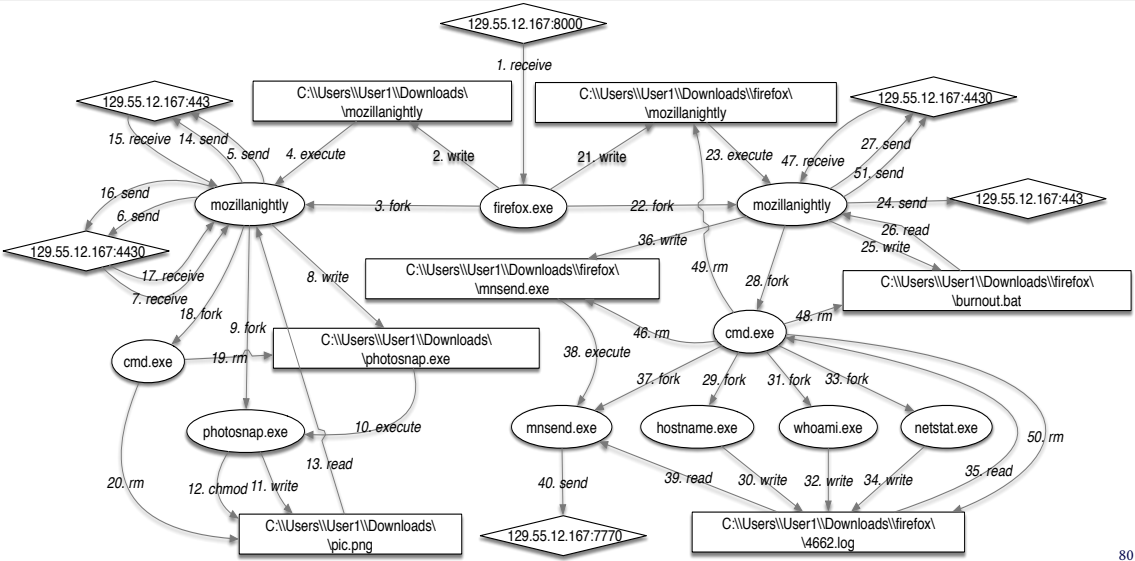


Forward Analysis: Key Ideas

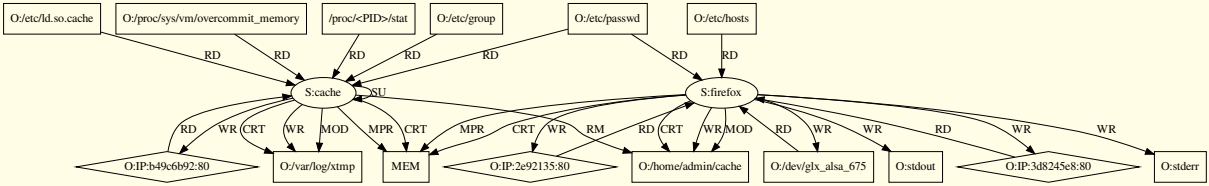
- Use cost metric to prune off distant nodes, i.e., nodes at a distance $\geq d_{th}$
- Cost metric favors
  - edges with untrusted *code* trustworthiness (cost=0);
  - and, to a lesser extent, edges with untrusted *data* trustworthiness (cost=1)



Example Campaign Graph Generated by SLEUTH

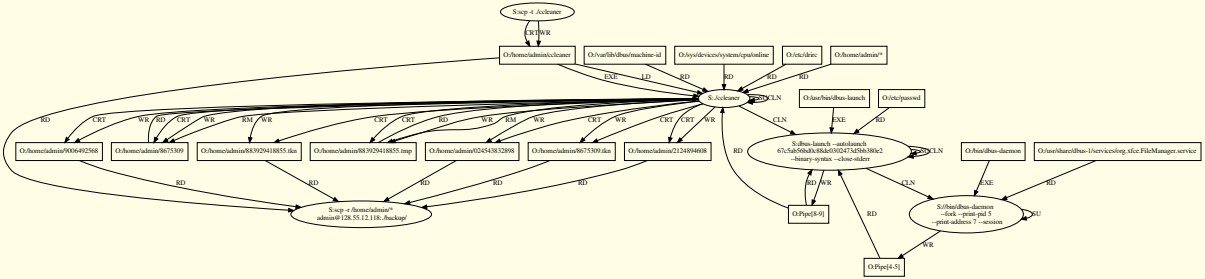


Graph for In-Memory Attack with Pre-existing Malware



- 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, laternal 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