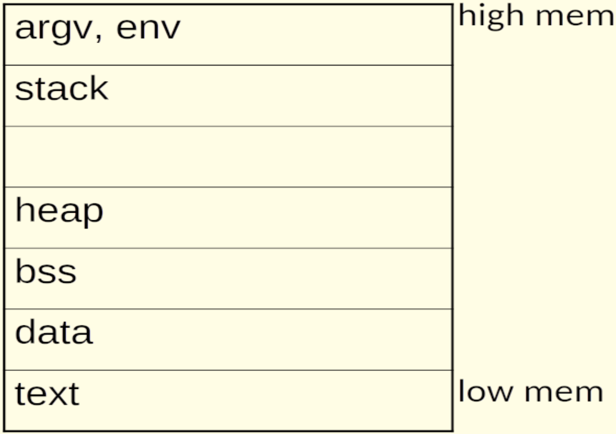
Memory Errors: Exploits and Defenses

Fall 2024

R. Sekar

Background: Process memory layout, Stack access and Calling conventions

Process Memory Layout



Argv/Env: Command-line 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

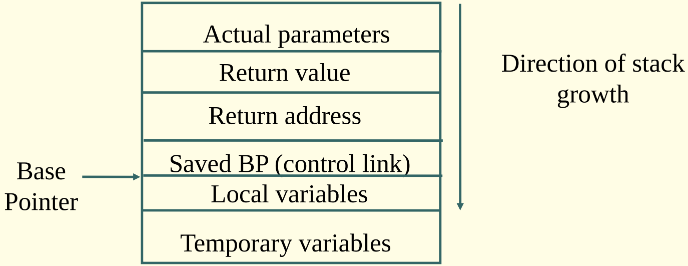
```
int a[] = {1, 2, 3, 4, 5}; //  DS: initialized global data
int b;                     // BSS: uninitialized global data

// text segment: contains program code
int main(int argc, char **argv) /* ptr to argv */ {
    int *c;                // stack: local variables

    c = (int *)malloc(5 * sizeof(int));
    // heap: dynamic allocation by new or malloc
}
```

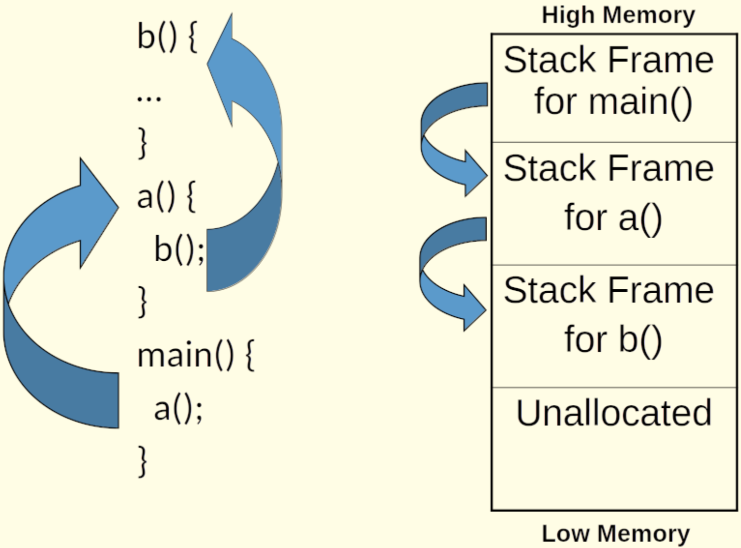
Call Stack and Activation Records

- An Activation Record (AR) — also called a *stack frame* — is created for each invocation of a procedure
- Structure of AR:



Base pointer is also called a *frame pointer*

Call Stack: Illustration



Accessing the Stack

Pushing an item onto the stack

- Decrement Stack Pointer by word size
  - 4 or 32-bit and 8 on 64-bit architectures
- Copy wordsize bytes of data to stack.
  - Example: `push 0x12`

Popping data from the stack

- Copy 4 bytes of data from stack.
- Increment SP by 4.
  - Example: `pop eax`

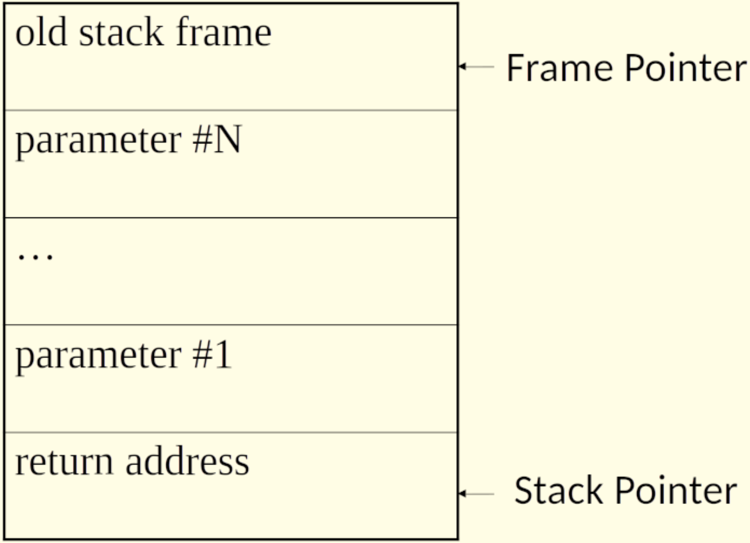
Stack Access

- Most items on the stack are accessed relative to Base pointer
  - Parameters
  - Local variables
  - Typically accessed using constant offsets hard-coded into the binary
- Register saves/restores typically use SP directly (push/pop)
- SP continually moves with push/pops.
- BP only moves on function call/return.
- Intel CPUs use ebp register for BP.
  - 32-bit registers are named eax, ebx, esp, etc.
  - 64-bit registers are named rax, rbx, rsp, etc.
- Optimized code can use SP for everything, freeing up BP for general-purpose use

C/ABI Calling Convention

- ABI: Application-binary interface
- Push all params onto stack in reverse order.
  - Parameter #N
  - ⋮
  - Parameter #2
  - Parameter #1
- Execute call instruction
  - Pushes address of next instruction (the return address) onto stack.
  - Modifies IP (eip) to point to start of function.

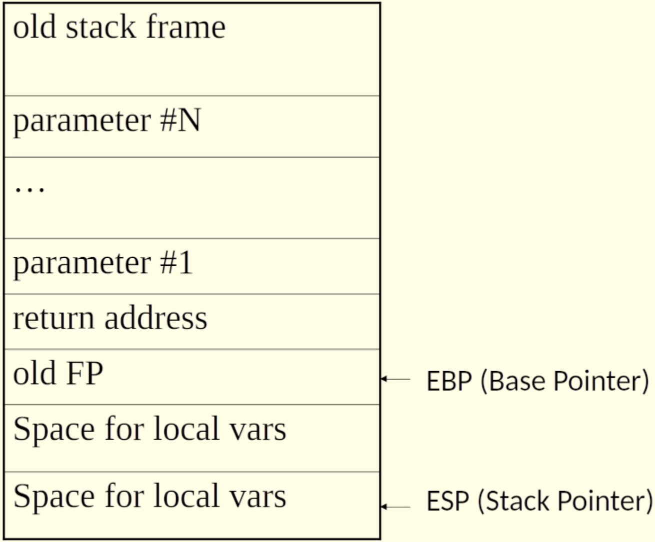
Stack just before the execution of callee



Callee's actions on the stack

- Function pushes BP (ebp) onto stack.
  - Save BP for previous function.
  - `push ebp`
- Copy SP to BP.
  - Allows function to access params as fixed indices from base pointer.
  - `mov ebp, esp`
- Reserves stack space for local vars.
  - `subl esp, 0x12`

Stack just before the execution of callee's body



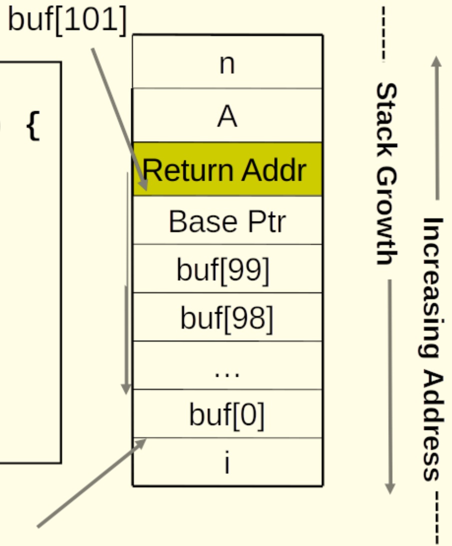
Callee's actions on the stack at its return

- Store return value in eax.
  - `movl eax, 0x0`
- Reset stack to pre-call state.
  - Destroys current stack frame, and restores caller's frame.
  - `mov esp, ebp; pop ebp`
- Return control back to the caller
  - `ret`
    - pops top word from stack and sets eip to that value.

Stack Smashing: Exploits, Defenses and Evasion Techniques

Stack Smashing Attack

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



Injected code starts here

Defense #1: Non-executable Data (aka *DEP*, *NX* or $W \oplus X$)

- Prevent execution of data
  - Programs very rarely need to do this
    - For programs that need to, create a more controlled interface, e.g., require another call to the OS `(mprotect) on Linux`
- Introduced in early 2000's
  - as soon as Intel added hardware support for this
- Counters direct code injection

Evasion #1.1: Use code already in victim process memory

- Often called "return-to-libc"
  - Because exploitable functions are there in `libc`, the low-level system library that is part of every program
- Examples
  - `system`: creates a shell to execute the argument (a string)
  - call `execve` syscall to run any program present on the victim
- Attacker needs to control the arguments to this victim function
  - Easy: attacker controls the stack contents, and the victim is getting its argument from the stack
- Typically, attacker will execute a shell, e.g., `/bin/bash`,
  - Attacker has full remote access on the victim now.

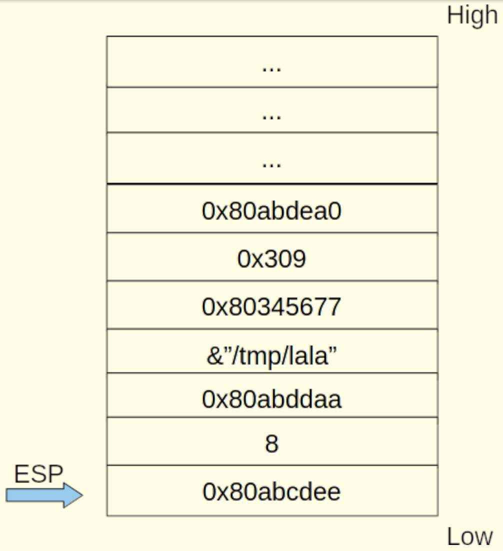
Evasion #1.2: Return-Oriented Programming (ROP)

- Why limit ourselves to just one or two functions?
  - What if the victim makes them hard to exploit?
- What happens if the argument is a pointer?
  - Attacker may not know the exact address where his/her data resides
- What if attacker needs a low-level primitive for which there is not an exploitable function?
  - Example: `mprotect` to make stack executable
  - Often, the end goal of attackers (so they can execute arbitrary code)

ROP: Realizing a Stack-based VM w/ Existing Instructions

- SP is the attacker's PC (program counter)
  - Points to "abstract instructions" on the stack
- Stack contains the attacker's "program"
- Victim's code serves like the attacker's data
  - Attacker picks the bytes in the victim code to use in the ROP payload
  - These bytes are called "gadgets"
  - ROP payload execution = execution of a series of gadgets on x86
- Variable-length x86 instruction plays a key role
  - Enables Turing-complete computation for most programs

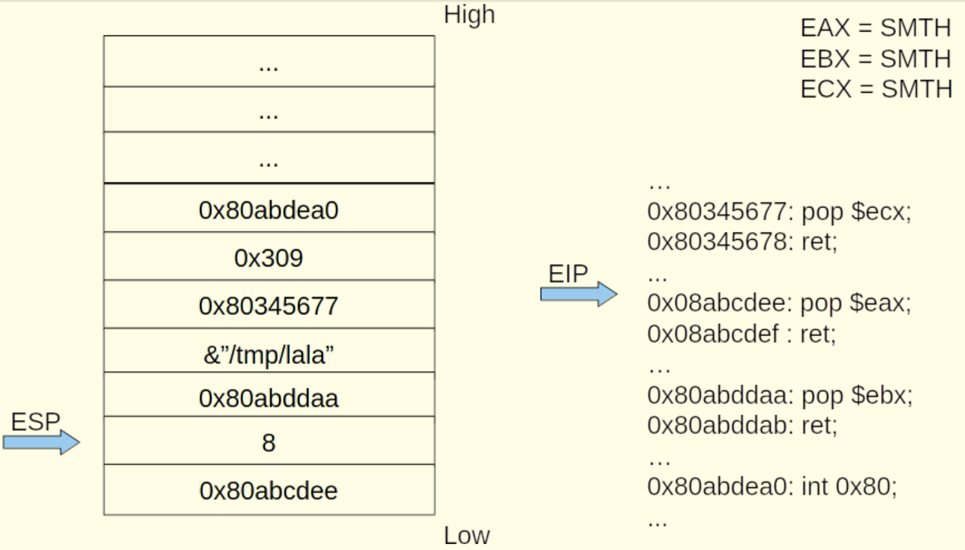
ROP Illustration



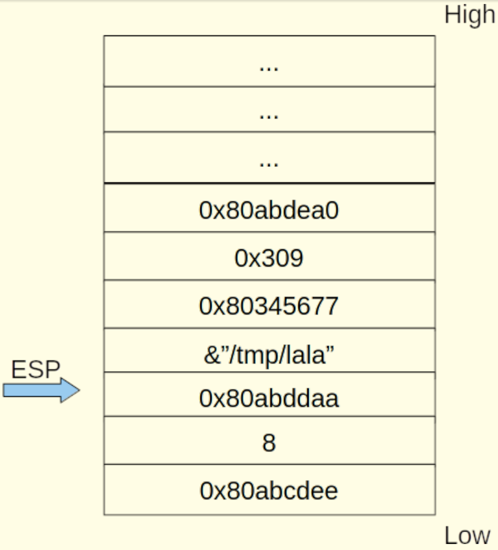
EAX = SMTH
EBX = SMTH
ECX = SMTH

...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...

ROP Illustration



ROP Illustration

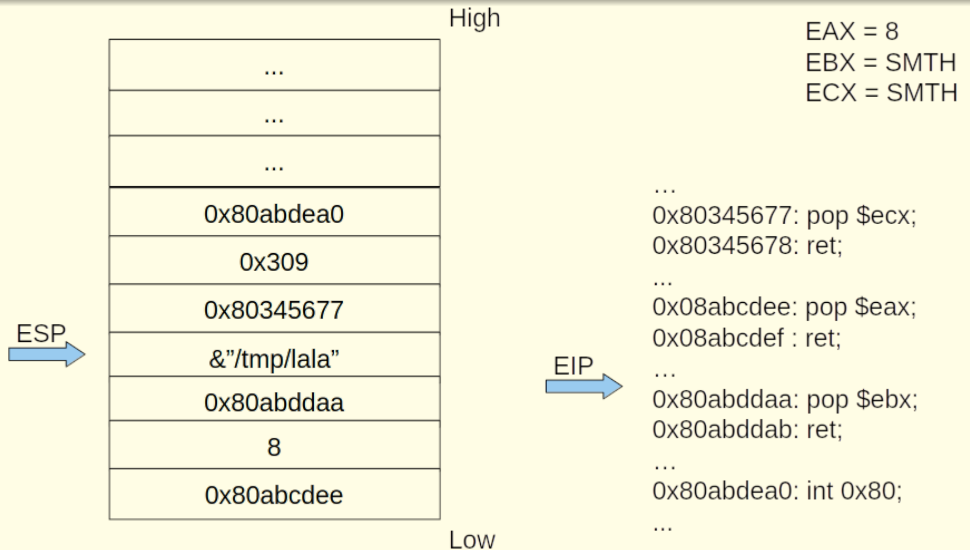


EAX = 8
EBX = SMTH
ECX = SMTH

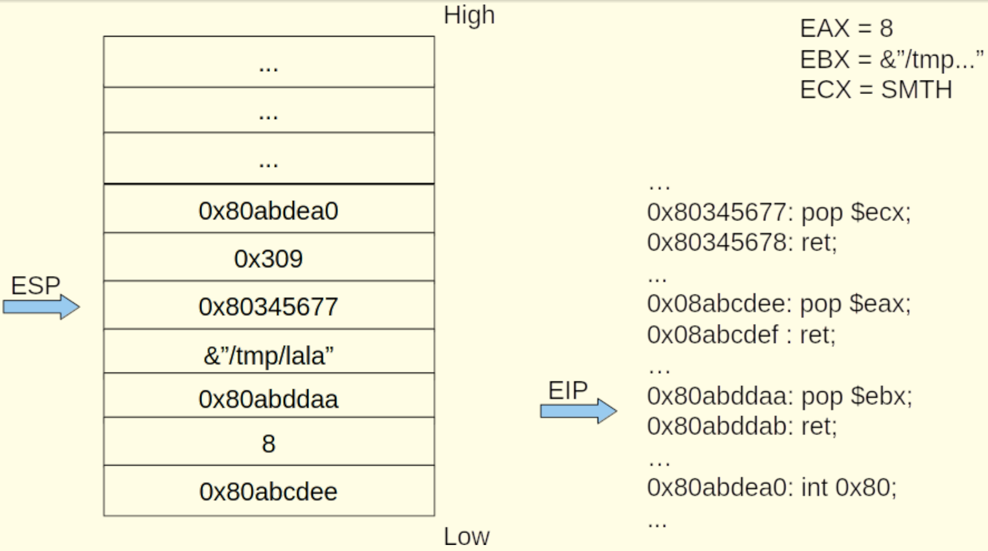
EIP

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

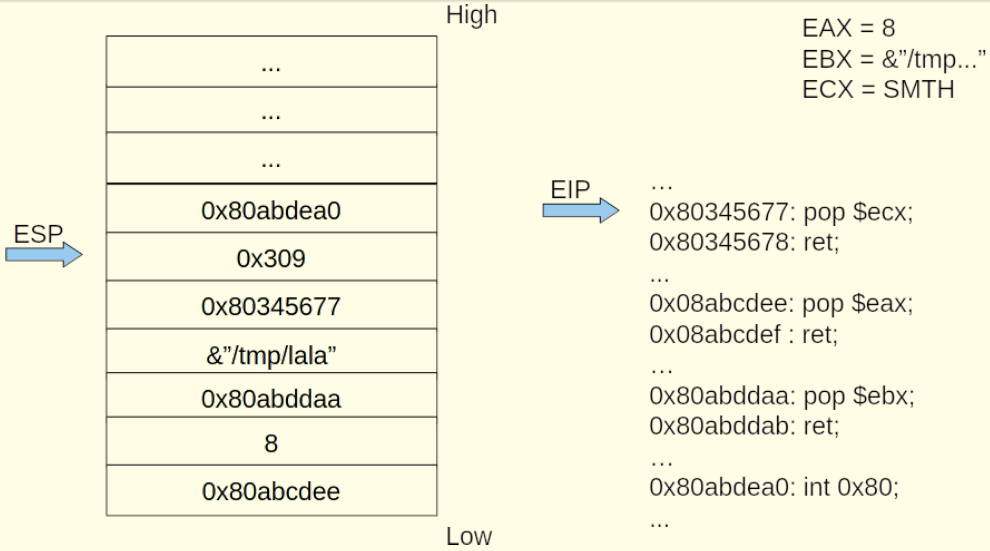
ROP Illustration



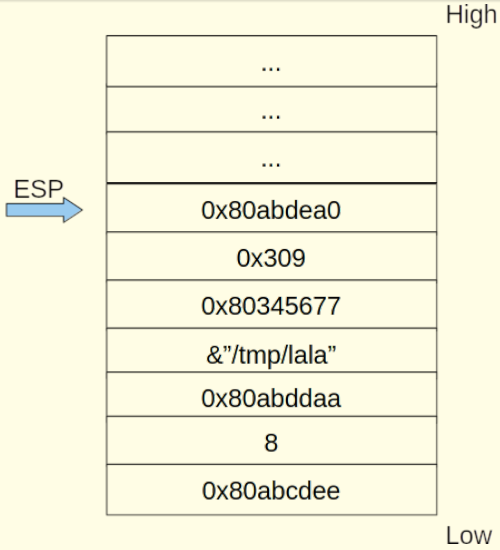
ROP Illustration



ROP Illustration



ROP Illustration

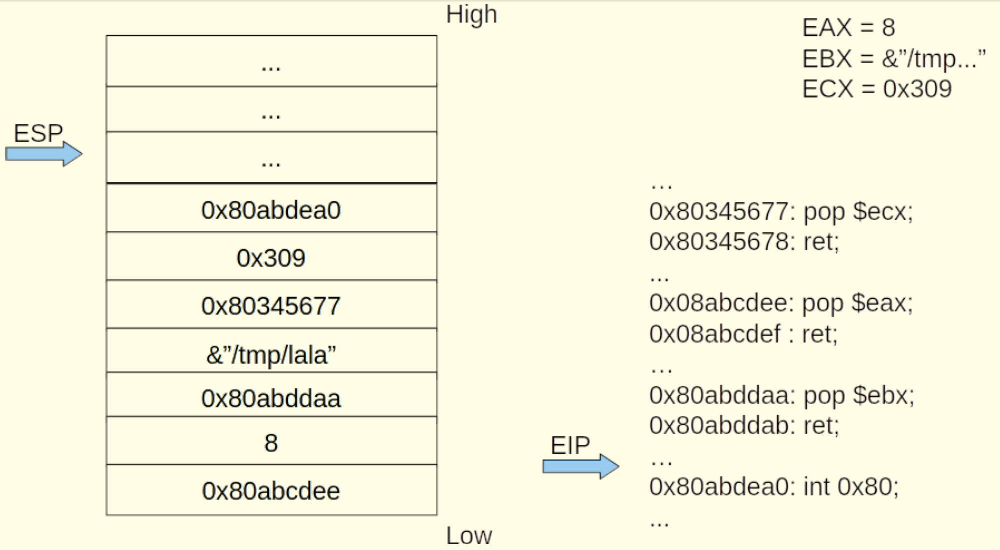


EAX = 8
EBX = &"/tmp..."
ECX = 0x309

EIP

...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...

ROP Illustration



Defense #2: Stack Canary

Store a "canary" value on the stack

- Callee generates and stores a canary value on function entry
- This value is checked at return
- If the canary is "dead" then abort the program
    - Indicates a stack overwrite
    - Turns control-flow hijack into DoS

Canary defense: Issues

- Fixed value vs random vs XOR
  - If the value is fixed and known in advance, the attacker can overwrite canary without being detected
    - Exception: Zero values and overflows due to `strcpy`
    - Can't preserve canary *and* overwrite RA
  - A random canary value seems harder for attacker
    - Information leakage: rely on a vulnerability that reveals canary value to the attacker
  - XOR canary avoids the need for an additional location
    - But breaks compatibility stack tracing and debuggers
- What is protected? RA?
  - What about Saved BP? Local variables?

ProPolice: Contemporary canary-based defense

old stack frame
parameter #N
...
parameter #1
RA
saved BP
Canary
Array-type local variables
Non-array type local variables

- Random canary value generated at process start time
- Protect BP by locating canary below saved BP
- Reorder local variables so that "simple" variables occur after variables subject to overflow.
  - Any overflow will go into canary, not the simple vars
- New on 64-bit
  - Make one byte of 64-bit canary into zero
  - Combines benefits of random and null canaries

Indirect (aka double-pointer) overwrite vulnerability

```
void parse_cmd(char* cmd) {
    char *arg = malloc(1024);
    char cmdnm[128];
    int i=0;
    while (!isspace(*cmd))   // Command name should end with
        cmdnm[i++] = *cmd++; // space; copy it into cmdnm
    cmd++;                   // Skip the space
    strcpy(arg, cmd);        // Copy rest of cmd into arg
    ...
    ...
    return
}
```

The exploit ...

...
parameter #1
RA
saved BP
Canary
char *arg
cmd[123..127]
...
cmd[0..3]

```
char *arg = malloc(1024);
char cmdnm[128];
int i=0;
while (!isspace(*cmd))
    cmdnm[i++] = *cmd++;
cmd++;
strcpy(arg, cmd);
```

- Overflow past `cmd[127]` to overwrite `arg` ...
  - ... so that it points to RA!
- Next, `strcpy` copies the rest of cmd into `arg`
  - Since `arg` points to RA, this operation overwrites RA!
- Canary is untouched!
  - But an XOR canary can still catch the attack

Brute-force attacks and Partial overwrites

- Brute-force attacks: try every possible value for canary until you succeed
  - With 32-bit canaries, this is feasible although it may take a while.
    - Requires victim process to restart *and use the same canary*
    - Use of same canary is not uncommon — forking-based servers
  - An attacker may also target millions of victims at once
    - Increases the probability that the canary is right for some victim
- Partial overwrite: guess the canary 1-byte at a time
  - Overwrite the first byte of canary
  - Repeat the attack for each possible value of byte
  - If victim did not crash, you got the right byte!
    - Now proceed to guess the next byte

Information Leaks

- Exploit a memory error that allows reading arbitrary memory location
  - Common example: format string attacks
  - When the victim contains `printf(s)` with `s` provided by attacker
  - `printf` blindly interprets stack contents as arguments
  - attacker may control the stack
- Partial overwrite is also an information leak ...
  - ... through a side-channel
  - An example of a *side-channel attack*

Other defenses for Return address

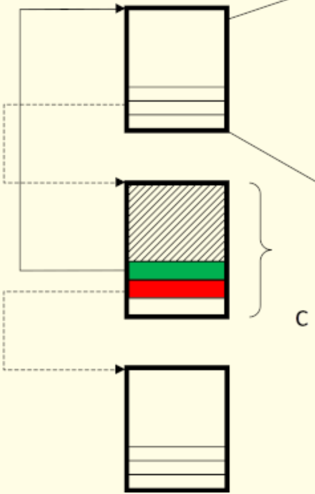
- Shadow Stack: Store a second copy of RA
  - Far more resilient than canaries
    - Bullet-proof if the second stack is unwritable
  - Causes compatibility issues if not implemented in every piece of code
  - Intel has built this capability into its processors
    - OSes and compiler tool chains need add to support (in progress)
- Safe Stack: no arrays of any kind on the stack
  - Already implemented into some compilers (LLVM) ...
  - ... if certain compiler flags are used

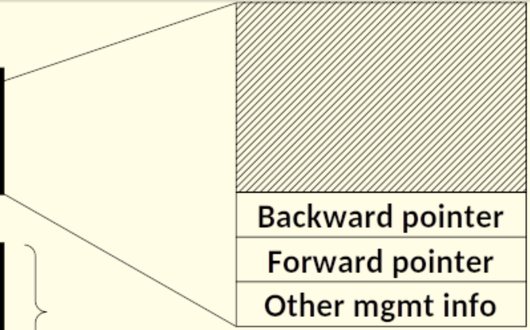
Beyond Stack-smashing: Heap overflows, Format string vulnerabilities, Integer overflows and Use-after-free

Overflows in Heap-allocated buffers

- 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 heap metadata
  - Overwriting a function pointer
    - Easiest target is a function pointer stored in the same heap block
    - C++ objects contain a built-in target: pointer to virtual function table
- Note: There may be other reasons for attackers to target code pointers$\neq$RA
  - e.g., if RA protection is very difficult to get around

How does heap metadata overwrite work?





Free chunk

Dlmalloc maintains a doubly linked list of free chunks

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

Or: red value is written 12 bytes after where green value points

```
c->backward->forward = c->forward
```

Heap metadata overwrite

- Provides a primitive to write an attacker-chosen value to an attacker chosen location
  - The ultimate capability sought by an attacker in a low-level exploit!
- Any doubly linked list implementation has this vulnerability!
  - Unless the program performs some kind of sanity checking.
  - This kind of sanity checking is implemented in malloc and other critical doubly linked lists.
  - But it is not always clear what to check
- Some systematic solutions
  - Heap canaries: protect heap metadata with canaries
  - separate metadata from data

Format-string vulnerabilities

- Exploits code of the form
  - ... read data from attacker into `s`...
  - `printf(s);`
- Printf usually reads memory, so how can it be used for arbitrary write?
  - "%n" primitive allows for a memory write
  - Writes the number of characters printed so far (character count)
- Primitive: write # of chars printed to an attacker-chosen location
  - Attacker typically controls the stack, so can choose the location written.
  - Only limited control over the value written, but most implementations allow just a single byte to be written
    - This is enough to easily control the value

Integer Overflows

- Can take multiple forms
  - Assignment between variables of different widths
  - Assignment between variables of different signs
  - Arithmetic overflows
- Can subvert bounds and size checks
  - Allocate a buffer smaller than needed
  - “Escape” bounds checks, e.g.,

    ```
    if (sz < n) memcpy(buf, src, sz);
    ```

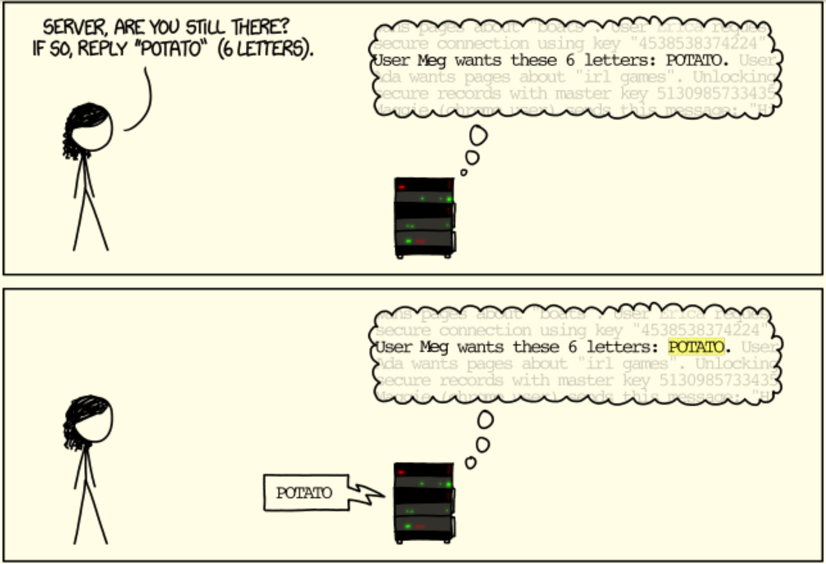
    A very large `sz` may become a negative integer!
- More info: `http://phrack.org/issues/60/10.html`

Use-after-free vulnerabilities

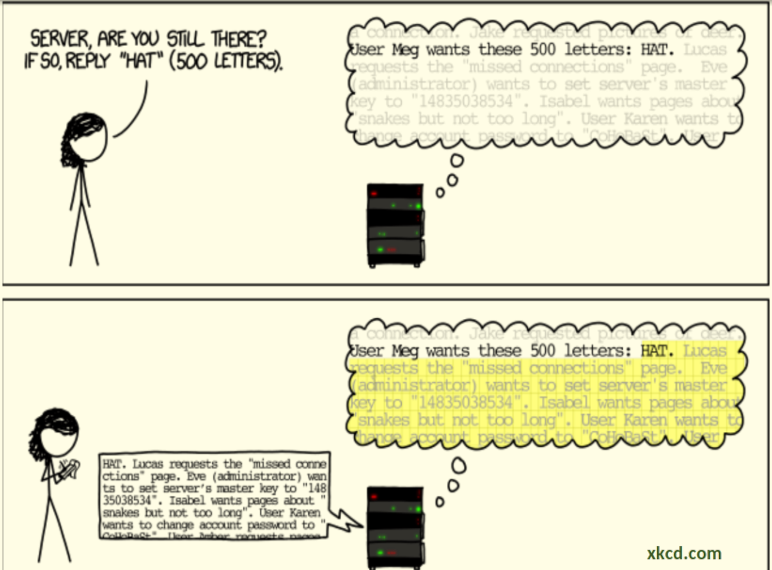
- Most past attacks were based on out-of-bounds writes
- But recently, attention has shifted to use-after-free
  - Access using dangling pointers
- Typical use in attacks
  - Victim uses a dangling pointer to access critical data
  - But the block is already freed and reallocated for processing (attacker's) input
- Can impact languages w/o pointers (PHP, Javascript)
  - if the bug is in memory managers of these languages

Overwrites are not the only serious problem

HOW THE HEARTBLEED BUG WORKS:



The Heartbleed Exploit!



Systematic Study of Memory Errors, Exploits and Defenses

Memory Errors

A memory error occurs when an object accessed using a pointer expression is different from the one intended by the programmer.

- Spatial error
  - 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)
  - dangling pointer errors
  - applicable to stack and heap allocated data

Use of Memory Errors in Attacks

- Most attacks used to be based on spatial errors, but in the last few years, temporal errors have become very important
  - "double free," "use-after-free"
- Typical attacks involve an out-of-bounds write (or a temporal error) to corrupt a pointer
  - This means that most attacks rely on multiple memory errors
    - 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.

Overview of Memory Error Defenses

- Prevent memory corruption
  - Detect and stop memory corruption before it happens
  - The most secure approach
    - but can have significant costs (performance and compatibility)
  - Techniques often focus on a subset of errors
    - But comprehensive techniques do exist
- Disrupt exploits
  - Unlike previous group of techniques, corruption is *not* stopped
  - "Guarding" solutions
    - detection may be delayed by a long period after corruption
    - not all instances of corruption may be detected
    - but can still seriously impair attacker's capabilities
  - Other disruption techniques impair control-flow hijack or payload execution

Preventing Memory Corruption

- Subclass of spatial errors: detect access past the end of valid objects
  - Introduce inter-object gaps, detect access to them (Red zones)
  - Purify, Light-weight bounds check [Hasabnis et al], Address Sanitizer [Serebryany et al]
- All spatial errors: detect by recognizing pointer arithmetic that crosses object boundaries
  - 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]
- Spatial and temporal errors
  - 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.

Disrupt exploits

1. Disrupt mechanism used for corruption
   - Protect attractive targets against common ways to corrupt them ("guarding" solutions)
2. Disrupt mechanism used for take-over
   - Disrupt ways in which the victim program uses corrupted data
   - Randomization-based defenses
3. Disrupt payload execution
   - Data execution prevention, Control-flow integrity (CFI), ...

(1) is highly incomplete, (3) is somewhat incomplete, so let us focus on (2).

Disrupt Take-over (Control-flow hijack)

- 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
    - Example: Stack-smashing, heap overflows, ...

Disrupting exploits: Diversity Based Defenses

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

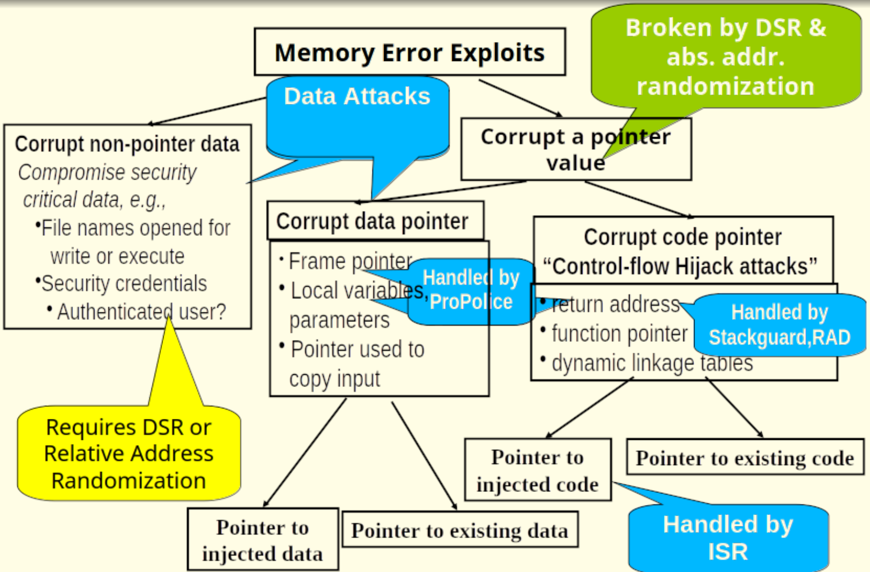
Automated Introduction of Diversity

- Use transformations that preserve program semantics
  - *How to capture intended program semantics?* Relying on manual specs isn't practical
- Approach: Focus on PL semantics, not the semantics of a specific program.
  - Randomize implementation aspects that aren't specified in the programming language
    - Eliminates need for programmer involvement
- 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 memory in a predictable way
  - Attacker knows the exact data item that is corrupted, e.g., RA.
    - Relative address randomization (RAR) takes away this predictability
  - Attacker knows the correct value to use for corruption, e.g., the location of injected code
    - Absolute address randomization (AAR) takes away this predictability for pointer-valued data
    - DSR takes away this predictability for all data

Space of Possible Memory Error Exploits



First Generation: Absolute Address Randomization (ASLR)

- Randomizes base address of:
  - data: stack, heap, static memory
  - code: libraries and executable regions
- Implemented on all mainstream OS distributions
  - On 32-bit systems, UNIX systems provide 20+ bits of randomness, 16 bits for Windows
  - 64-bit systems add about 16 additional bits of randomness.
- Limitations
  - Incomplete implementations (e.g., executables or some libraries left unrandomized)
    - but this is becoming rare these days.
  - Brute-force as well as smarter guessing attacks (e.g., partial overwrites)
  - Brute-force in space domain: NOP padding, *Heap spray*
  - Information leakage attacks
  - Relative address data-only attacks

Second Generation: Relative Address Randomization

- Randomize distance between static objects
  - Compile time
    - Often, linking time
  - Load time: Requires additional information in binaries
- Randomize distance between stack objects
  - Entropy is limited if the number of variables is small
  - Better option: safe stack for simple variables, move rest to heap
- Heap allocations can be randomized without help from compiler.

Fine-grained code randomization (RAR for code)

- Motivation: make ROP infeasible
  - Permute order of functions
  - Randomly rearrange instructions within a function
- Attacker response
  - Just-in-time ROP
  - Blind ROP

Benefits of RAR

- Defeats the overwrite step, as well the step that uses the overwritten pointer value
  - Can mitigate format-string and integer overflow attacks as well
- Provides higher entropy
- Unlike AAR, a single information leak insufficient to derandomize everything.
  - Knowing the location of one object does not tell you much about the locations of other objects

Data Space Randomization

- Basic idea: Randomize data representation
  - Xor each data object with a distinct random mask
  - Effect of data corruption becomes non-deterministic
    - Out-of-bounds access on array a to corrupt variable x with value v
    - Actual value written is $mask(\mathrm{a}) \oplus v$
    - Value read when x is accessed: $mask(\mathrm{x}) \oplus (mask(a) \oplus v)$ — random gibberish
- Unlike AAR, protects all data, not just pointers
- Effective against relative address as well as absolute address attacks
- Large entropy
- Key challenge: Requires alias analysis
  - Objects that may be pointed by the same pointer must use same mask