CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Buffer Overflow Defenses

Fall 2017

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Admin

- Please make sure you can access Lab 1 asap!
- Reminder: Lab 1 is much easier if you do the recommended reading (see course schedule for links):
  - Smashing the Stack for Fun and Profit
  - Exploiting Format String Vulnerabilities

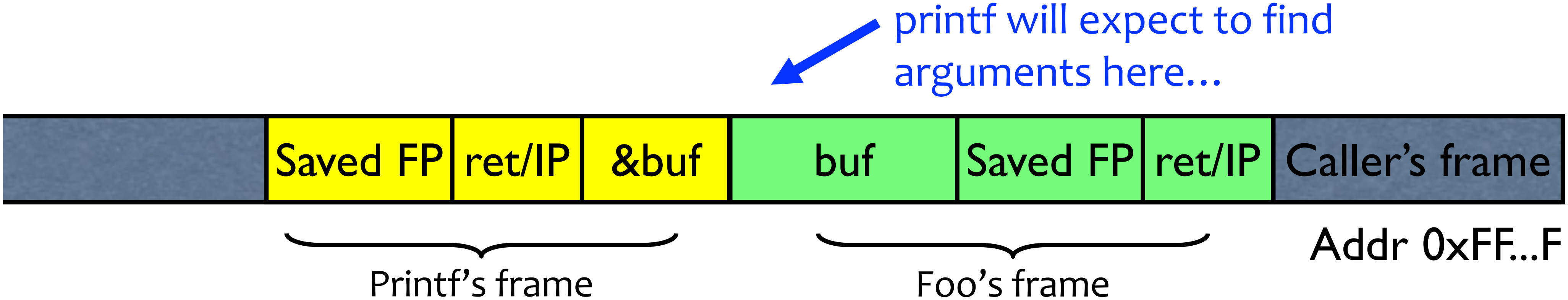
Reminder: Printf

- Printf takes a variable number of arguments
  - E.g., printf("Here's an int: %d", 10);
- Assumptions about input can lead to trouble
  - E.g., printf(buf) when buf="Hello world" versus when buf="Hello world %d"
  - Can be used to advance printf's internal stack pointer
  - Can read memory
    - E.g., printf("%x") will print in hex format whatever printf's internal stack pointer is pointing to at the time
  - Can write memory
    - E.g., printf("Hello%n"); will write "5" to the memory location specified by whatever printf's internal SP is pointing to at the time

How Can We Attack This?

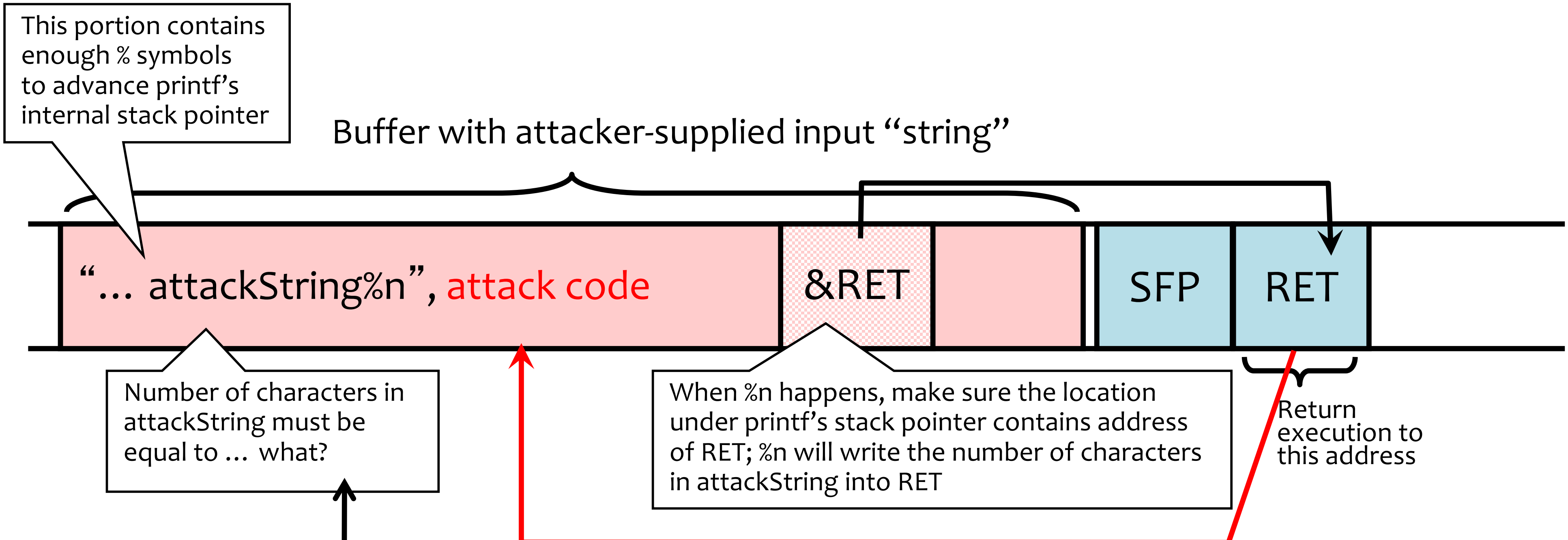
```
foo() {
        char buf[…];
        strncpy(buf, readUntrustedInput(), sizeof(buf));
        printf(buf); //vulnerable
}
```

If format string contains % then printf will expect to find arguments here…



What should readUntrustedInput() return??

Using %n to Overwrite Return Address



C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf("%5d", 10) will print three spaces followed by the integer: "   10"
That is, %n will print 5, not 2.

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.
(4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it

- Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack "canaries"
  3. Encrypt pointers
  4. Address space layout randomization

W-xor-X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows (since XP SP2), Linux (via PaX patches), OS X (since 10.5)

What Does W-xor-X Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap
- **As long as "saved EIP" points into existing code, W-xor-X protection will not block control transfer**
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code

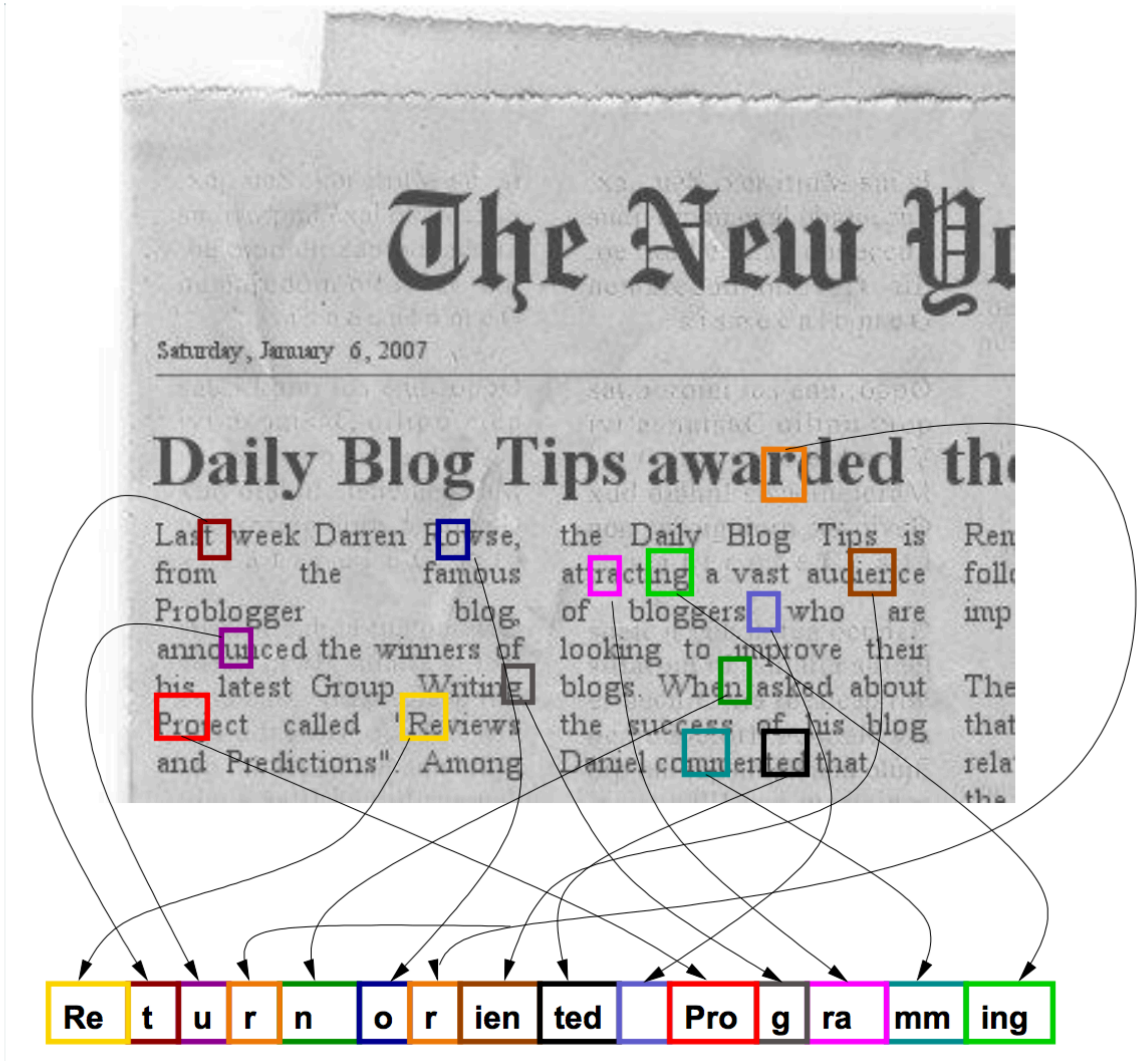
return-to-libc on Steroids

- 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!
  - 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

Chaining RETs for Fun and Profit

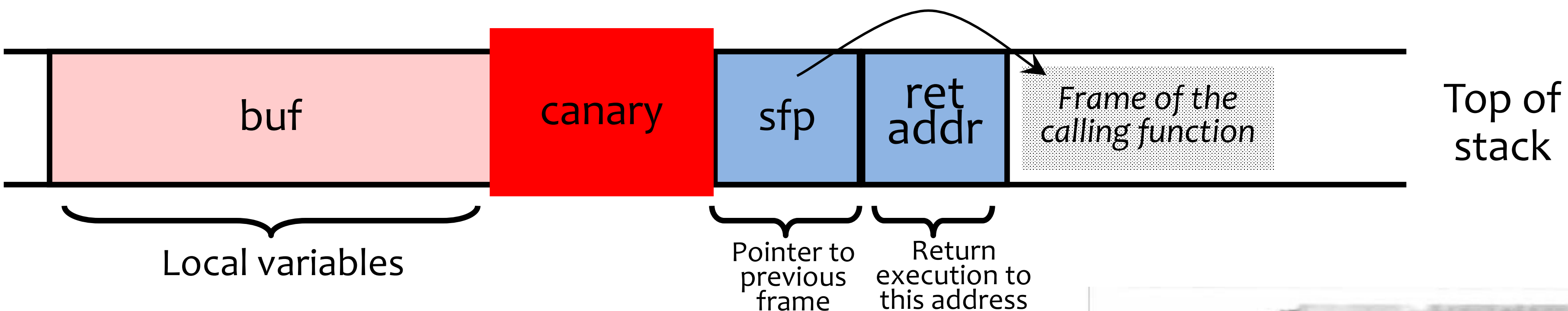
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming

Return-Oriented Programming



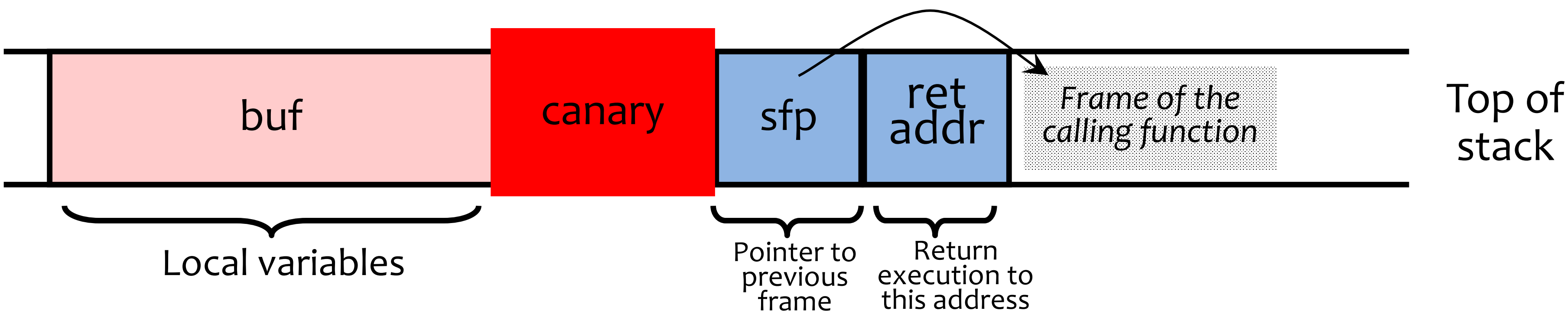
Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



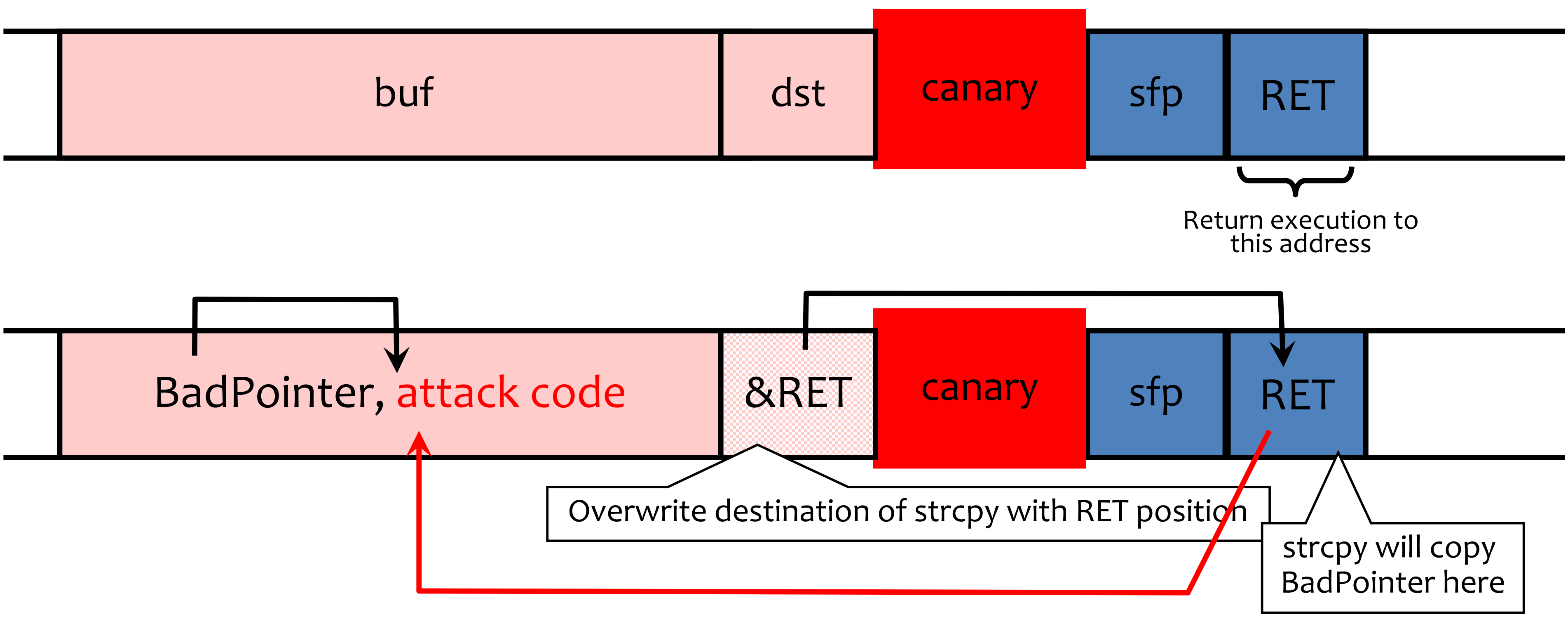
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

Defeating StackGuard

- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable

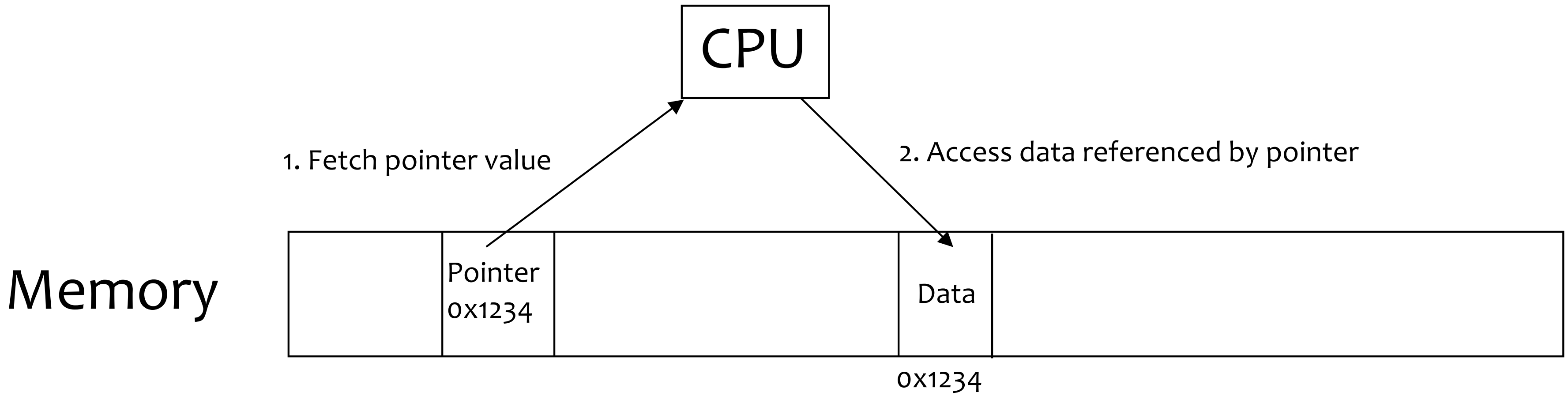


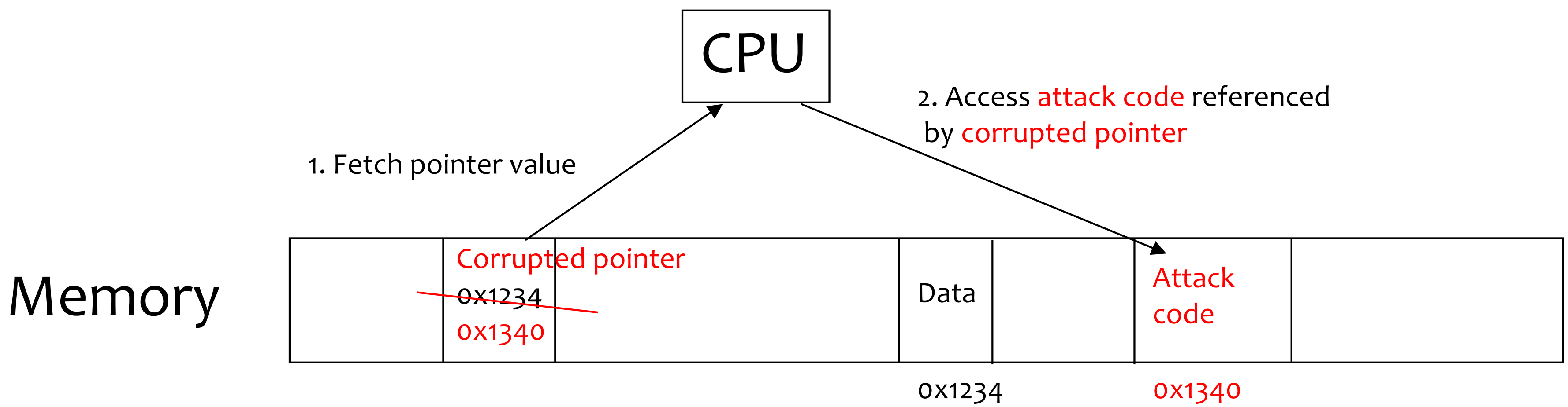
PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

[Cowan]

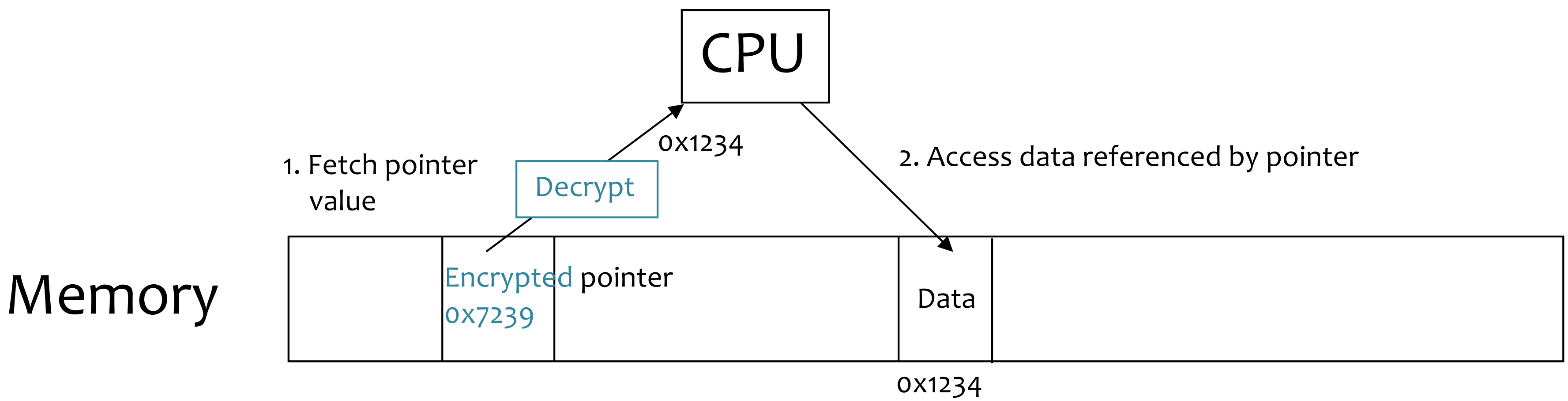
Normal Pointer Dereference

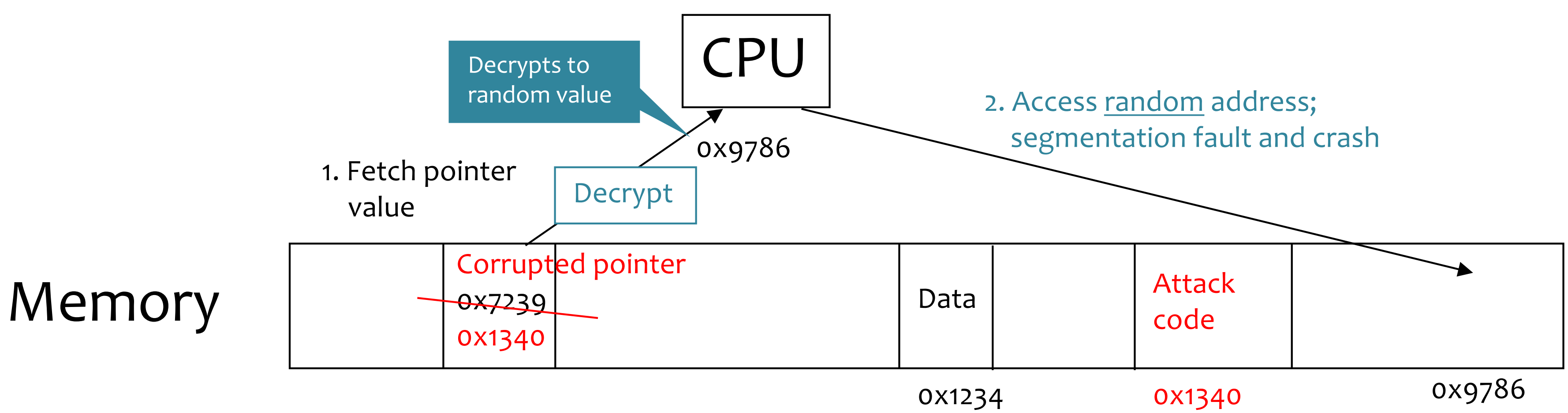




[Cowan]

PointGuard Dereference





PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt only pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

ASLR: Address Space Randomization

- Map shared libraries to a random location in process memory
  - Attacker does not know addresses of executable code
- Deployment (examples)
  - Windows Vista: 8 bits of randomness for DLLs
  - Linux (via PaX): 16 bits of randomness for libraries
  - Even Android
  - More effective on 64-bit architectures
- Other randomization methods
  - Randomize system call ids or instruction set

Example: ASLR in Vista

- Booting Vista twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

ASLR Issues

- NOP slides and heap spraying to increase likelihood for custom code (e.g., on heap)
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library

Other Possible Solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
  - (Though Java doesn't magically fix all security issues ☺)
- Static analysis of source code to find overflows
- Dynamic testing: "fuzzing"
- LibSafe: dynamically loaded library that intercepts calls to unsafe C functions and checks that there's enough space before doing copies
  - Also doesn't prevent everything