

CSEP 564: Computer Security and Privacy

Software Security: Buffer Overflow Defenses

Fall 2022


also
attacks

David Kohlbrenner


dkohlbre@cs

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

A note on summaries

- Please follow the instructions!
 - Don't just use bullets 
- We're looking for evidence you read the paper carefully and thought about it
 - Might have been harsh on this round, grades posting after class
 - Please read our feedback!

Lab 1

- Downtime resolved? 
- You should have started by now
 - Hopefully deep into, or solved, exploits 1+2
 - If exploit 4 didn't make sense, today should help 😊

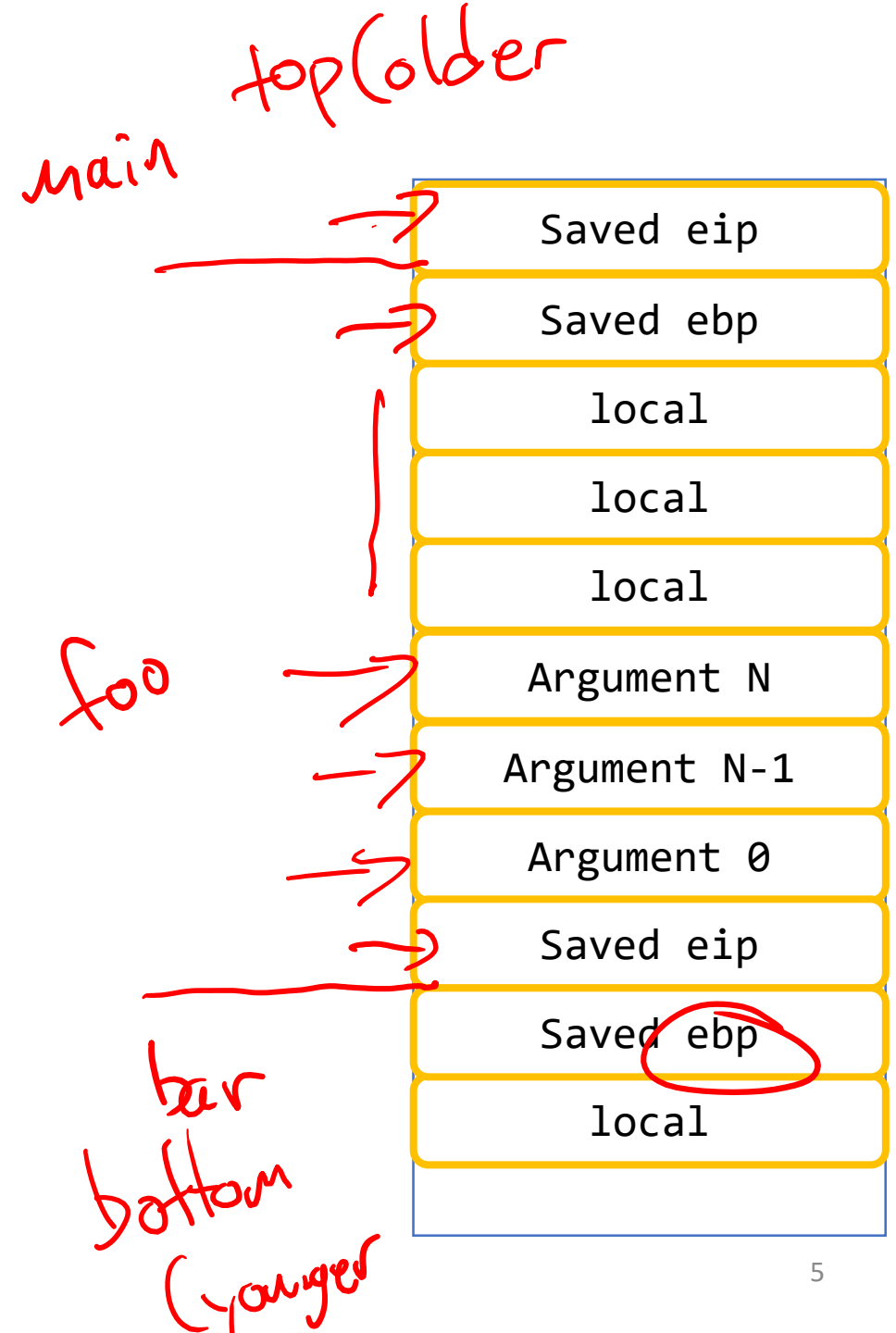
Lab 1

- Office hours were pretty packed, good!
- Common questions:
 - Around function preludes / epilogues
 - I can link some animations/tutorials on ed, none are perfect
 - Around what you can do with the sploit3 1-byte overwrite



Lab 1





- Office hours were pretty packed, good!
- Common questions:
 - Around function preludes / epilogues
 - I can link some animations/tutorials on ed, none are perfect
 - Around what you can do with the sploit3 1-byte overwrite



Paper Discussion Time!

“Automated Whitebox Fuzz Testing”

Patrice Godefroid, Michael Y. Levin, and David Molnar

- Pick one of these and ask about it/describe it to your neighbor briefly
 - Symbolic execution 
 - Whitebox vs blackbox fuzzing 
 - Path constraints 
 - Code coverage 
 - Something else from the paper

Paper Discussion Time!

“Automated Whitebox Fuzz Testing”

Patrice Godefroid, Michael Y. Levin, and David Molnar

- What is something you *learned* from this paper?

Last time...

- Stack smashing and overwriting return pointers
- “Computing” with printf



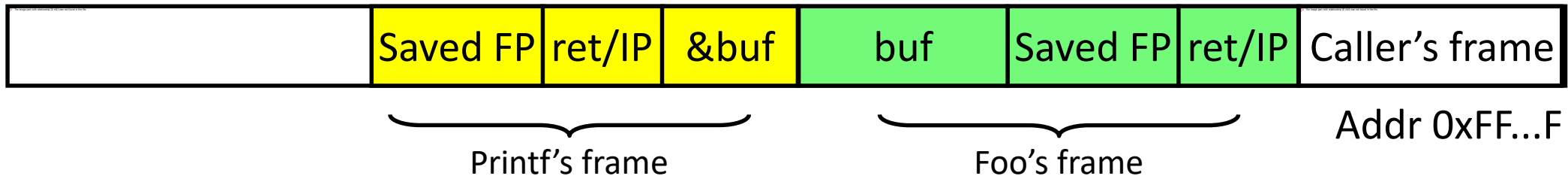
Summary of Printf Risks

- 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 the string returned by readUntrustedInput() contain??

Different compilers /
compiler options /
architectures might vary


The exploitation twilight zone

- During an exploitation attempt sometimes you have to ‘let it run’
 - Overflow a buffer
 - Change things
 - Let program run for ‘a bit’ 
 - Everything triggers! 
- Printf exploit a perfect example

Recommended Reading

- It will be hard to do Lab 1 without:
 - Reading (see assignments):
 - Smashing the Stack for Fun and Profit
 - Exploiting Format String Vulnerabilities

Buffer Overflow: Causes and Cures

- Classical 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
 5. Code analysis
 6. Better interfaces 
 7. ...

Defense: Better string functions!


$\text{len}(\text{src}) > \text{len}(\text{dst})$

- strcpy is bad
- strncpy is... also bad (no null terminator! Returns dest!)

Defense: Better string functions!

- strcpy is bad
- strncpy is... also bad (no null terminator! Returns dest!)
- BSD to the rescue: strlcpy
 - size_t strlcpy(char *dest, const char *src, size_t n);
 - Always NUL terminates
 - Returns len(src) ...

strncpy – maybe not what we wanted

- How do you check truncation?
- Endless arguments, no glibc implementation (!)
- Programmers instead do this:
 - `#define strncpy(dest,src,len) strncpy(dest,src,(len)-1)` 

Pollev/discussion

- What would you want a C string function to do from a safety perspective?
- Remember: a C string is an array of bytes terminated with a NUL byte.
- There are no other properties!



strscpy – Maybe this one is good

- `ssize_t strscpy(char *dest, const char *src, size_t count);`
 - NUL terminates no matter what
 - Returns `len(src)`

Should I even care? C string functions? Really?

Should I even care? C string functions? Really?

- <https://lwn.net/Articles/905777/>

Ushering out `strncpy()`

By **Jonathan Corbet**
August 25, 2022

With all of the complex problems that must be solved in the kernel, one might think that copying a string would draw little attention. Even with the hazards that C strings present, simply moving some bytes should not be all that hard. But string-copy functions have been a frequent subject of debate over the years, with different variants being in fashion at times. Now it seems that the BSD-derived `strncpy()` function may finally be on its way out of the kernel.

Defense: Executable Space Protection

- Mark all writeable memory locations as non-executable NX
 - Example: Microsoft's Data Execution Prevention (DEP)
 - This blocks many code injection exploits
- Hardware support
 - AMD "NX" bit (no-execute), Intel "XD" bit (executed disable) (in post-2004 CPUs)
 - Makes memory page non-executable
- Widely deployed
 - Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+

Pollev

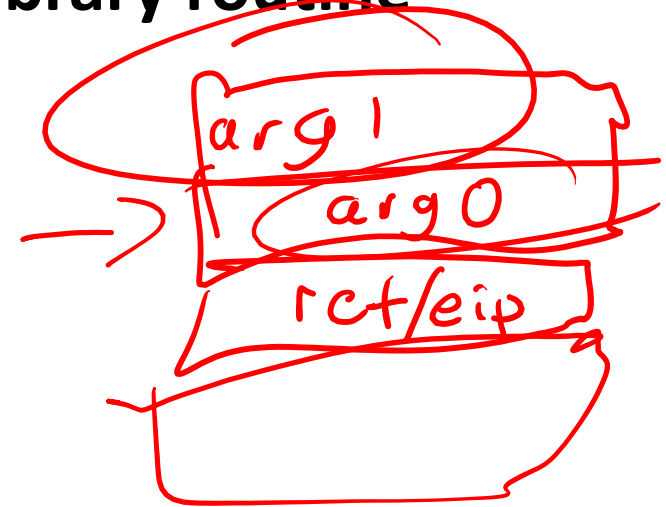
- What might an attacker be able to accomplish even if they cannot execute code on the stack?

What Does “Executable Space Protection” Not Prevent?

- Can still corrupt stack ...
 - ... or function pointers
 - ... or critical data on the heap
- **As long as RET points into existing code, executable space protection will not block control transfer!**
 - return-to-libc exploits

return-to-libc

- Overwrite saved ret (IP) with address of **any library routine**
 - Arrange stack to look like arguments
- Does not look like a huge threat
 - ...
- ~~Canvas in-class activity, Oct 8!~~



return-to-libc

- Overwrite saved ret (IP) with address of **any library routine**
 - Arrange stack to look like arguments
- Does not look like a huge threat
 - ...
 - We can call *any* function we want!
 - Say, exec 😊

return-to-libc++

- Insight: 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 (SP)
 - Guess what? Its value is under attacker's control!
 - Use it as the new value for IP
 - Now control is transferred to an address of attacker's choice!
 - Increment SP to point to the next word on the stack

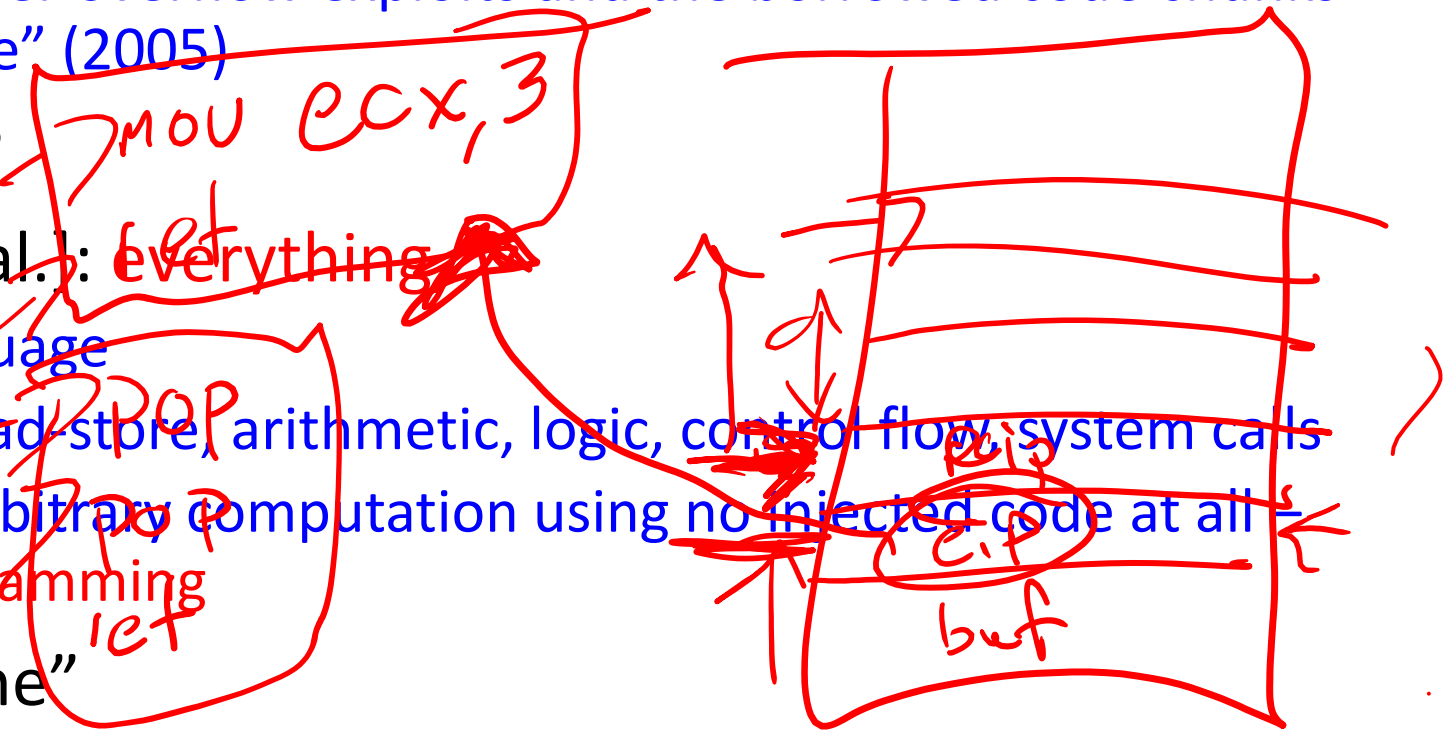


A hand-drawn diagram in red ink representing a gadget. It is a rounded rectangle containing the text "add", "mov", and "ret" stacked vertically. Below the rectangle, the text "gadget" is written in red. A red arrow points from the right side of the diagram towards the right edge of the slide.

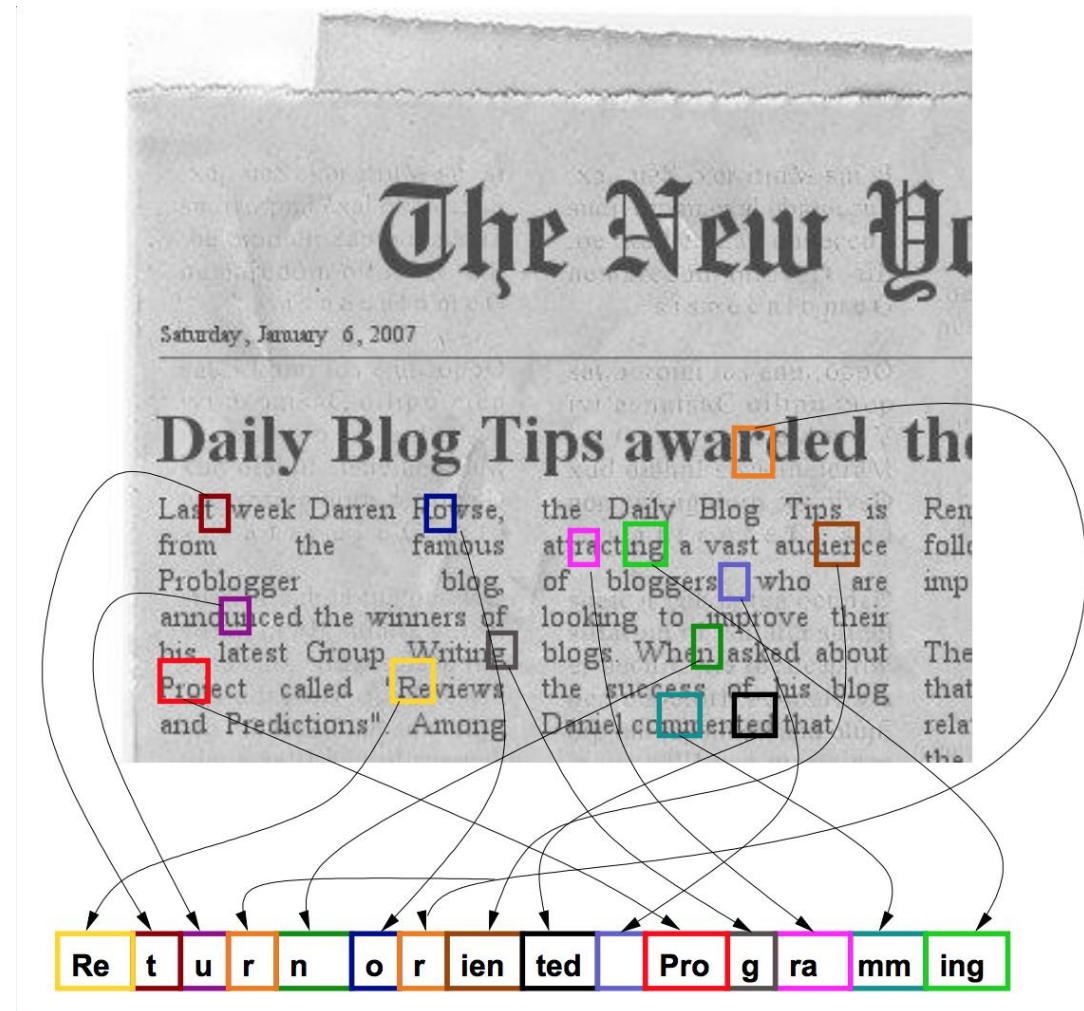
Chaining RETs

ROP JOP
~~ROP~~

- 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
- Truly, a “weird machine”

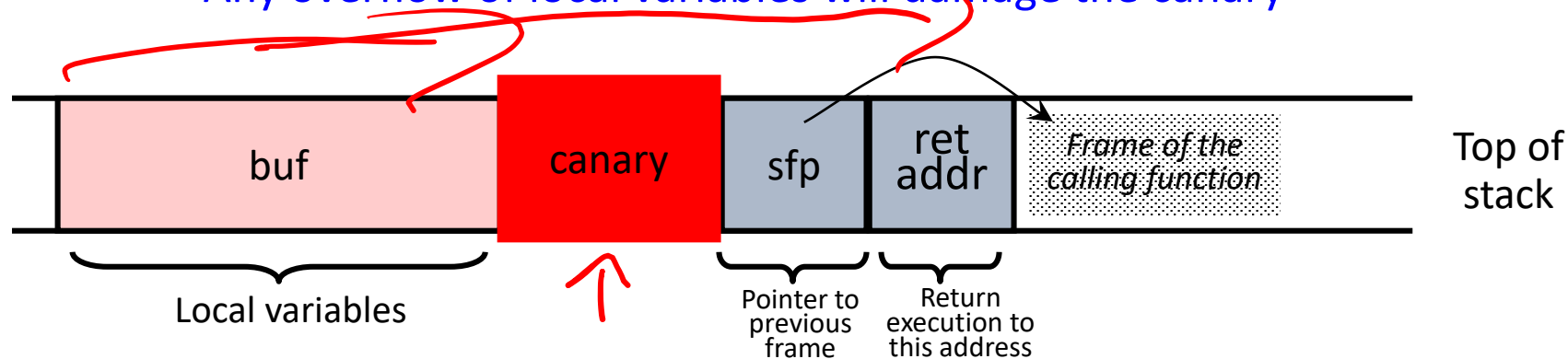


Return-Oriented Programming



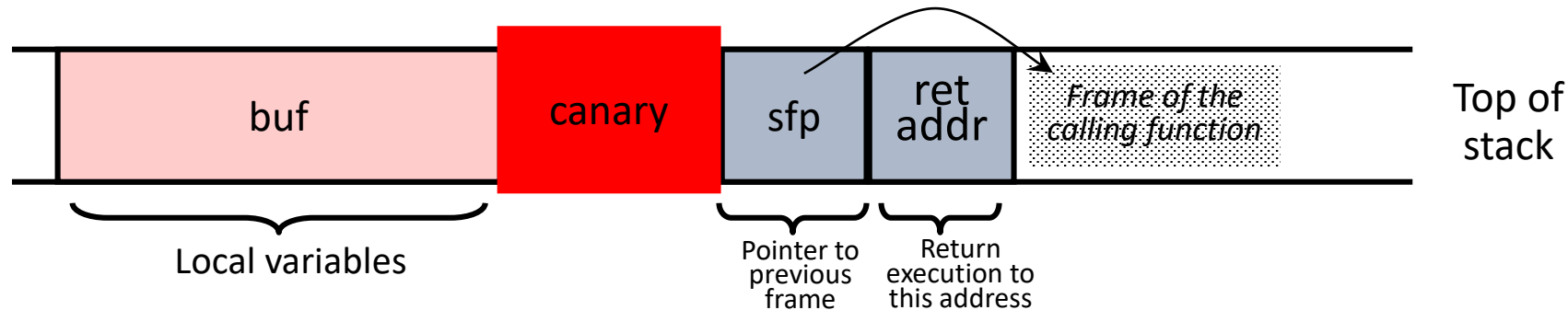
Defense: Run-Time Checking: StackGuard


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



Defense: 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



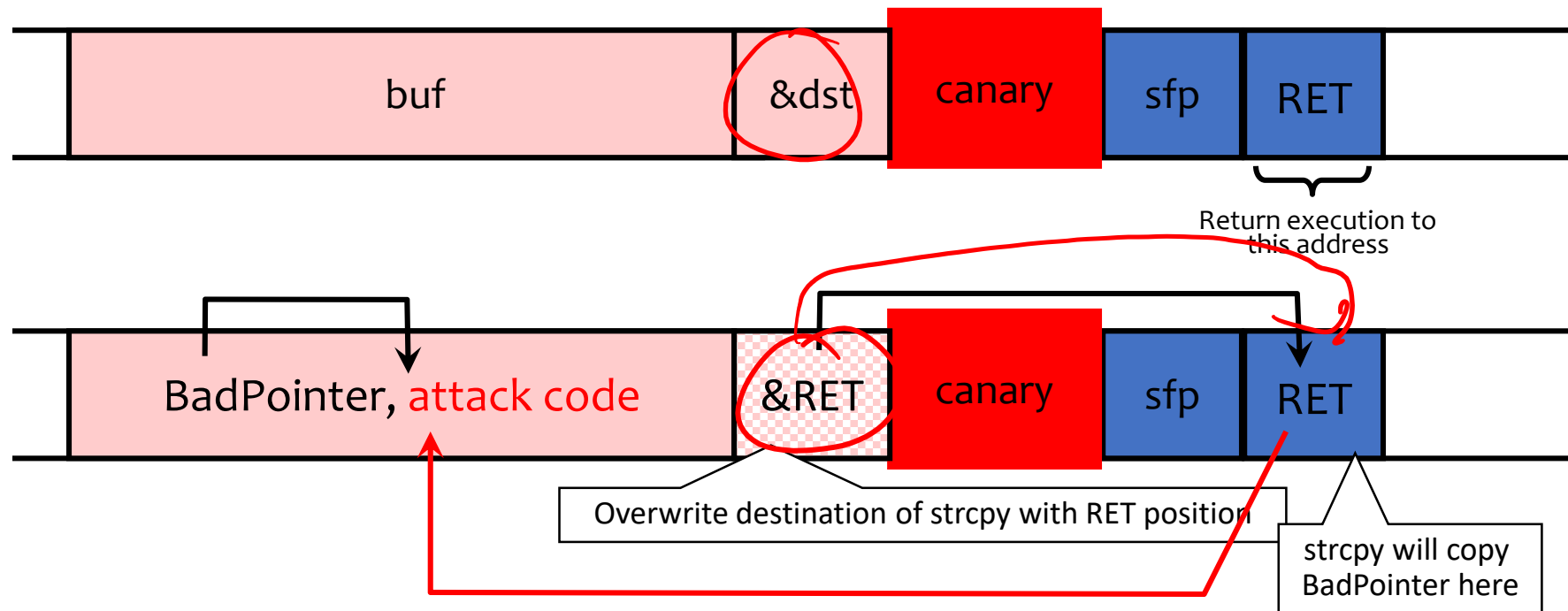
- Choose random canary string on program start
 - Attacker can't guess what the value of canary will be
- Canary contains: “\0”, newline, linefeed, EOF 
 - String functions like strcpy won't copy beyond “\0”

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 at one point in time

Defeating StackGuard

- StackGuard can be defeated
 - A single memory write where the attacker controls both the value and the destination is sufficient
- Suppose program contains `copy(buf,attacker-input)` and `copy(dst,buf)`
 - Example: dst is a local pointer variable
 - Attacker controls both buf and dst

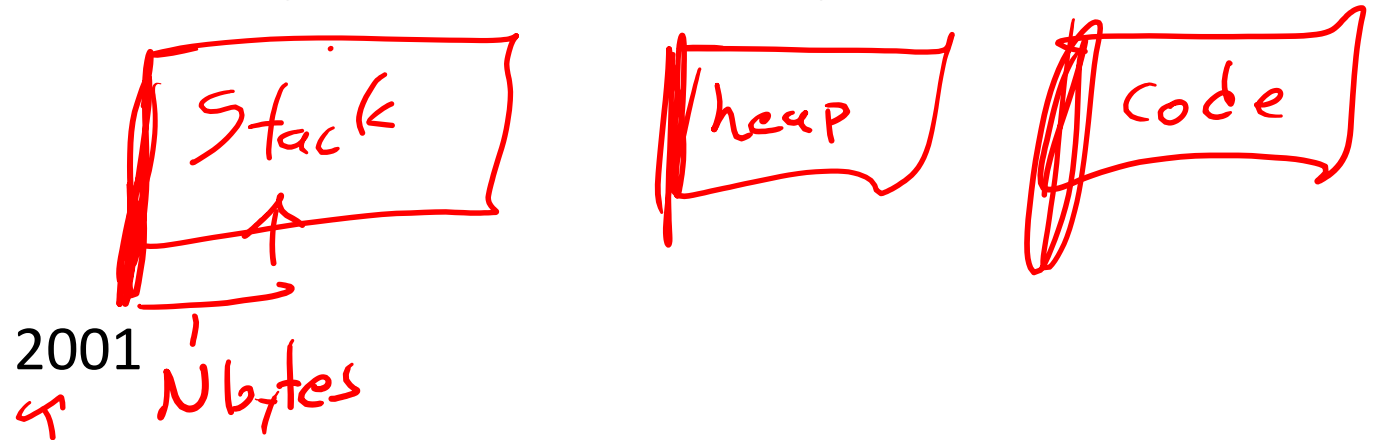


ASLR: Address Space Randomization

- Randomly arrange address space of key data areas for a process

- Base of executable region
- Position of stack
- Position of heap
- Position of libraries

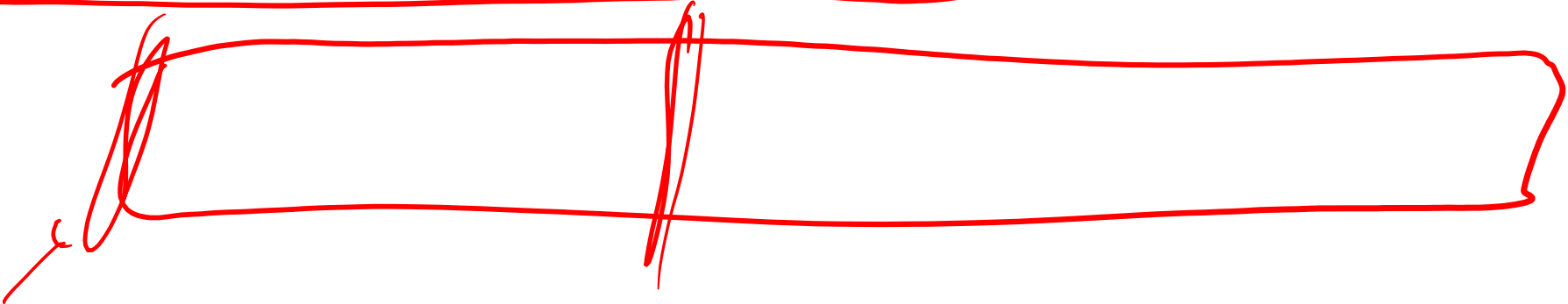
- Introduced by Linux PaX project in 2001
- Adopted by OpenBSD in 2003
- Adopted by Linux in 2005



PIC
PIE

ASLR: Address Space Randomization

- Deployment (examples)
 - Linux kernel since 2.6.12 (2005+)
 - Android 4.0+
 - iOS 4.3+ ; OS X 10.5+
 - Microsoft since Windows Vista (2007)
- Attacker goal: Guess or figure out target address (or addresses)
- ASLR more effective on 64-bit architectures

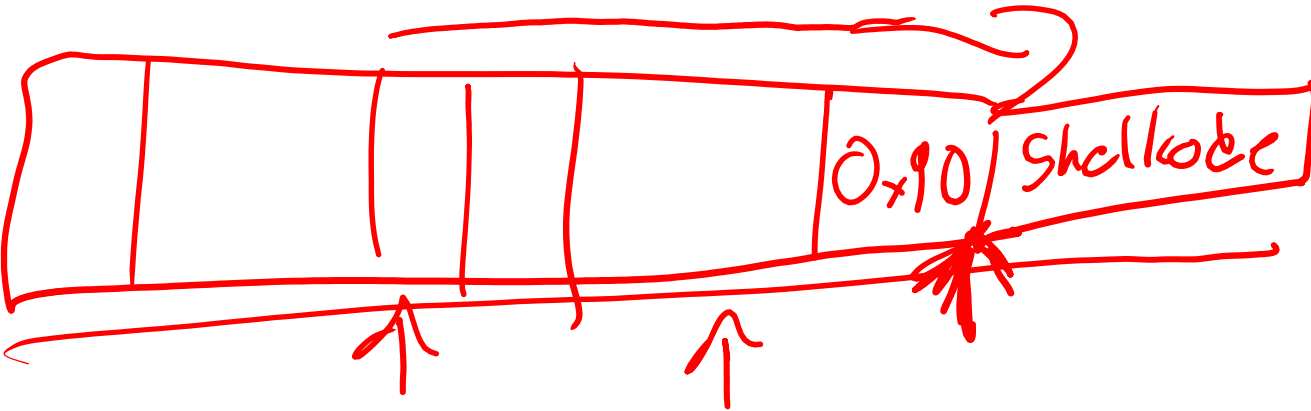


Attacking ASLR

- ➔ • **NOP sleds and heap spraying** to increase likelihood for adversary's code to be reached (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, depending on the ASLR implementation

Aslide: nopsleds

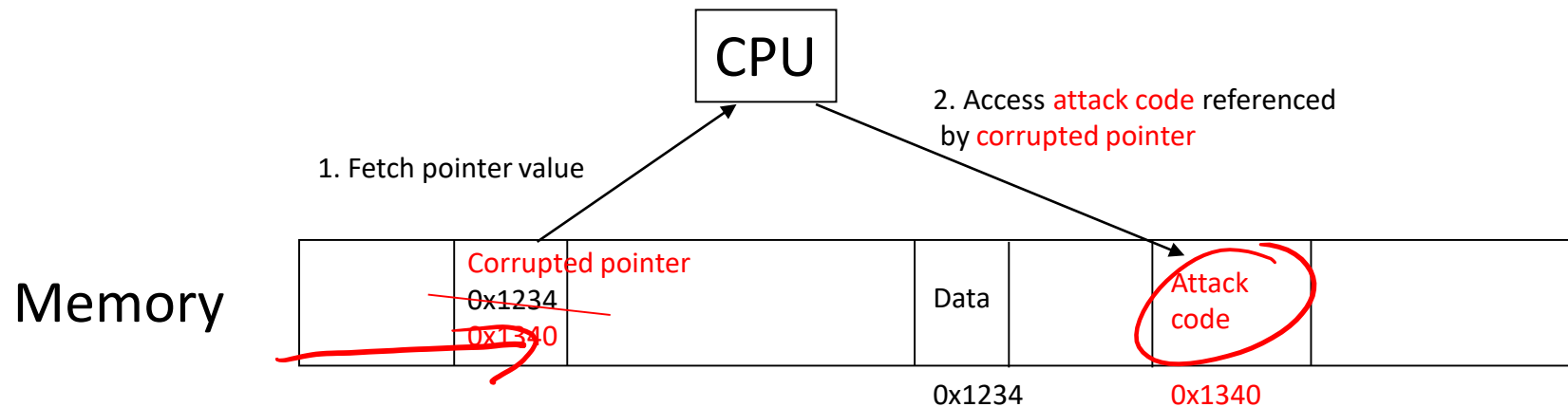
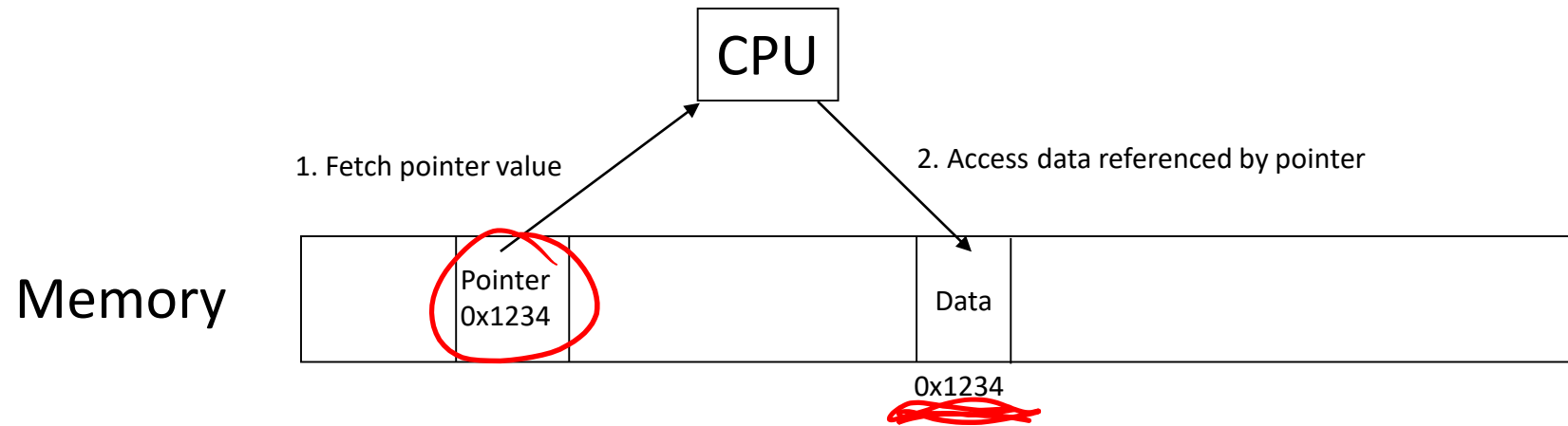
Pretend you can corrupt a saved return address, but you don't know where to point it to!



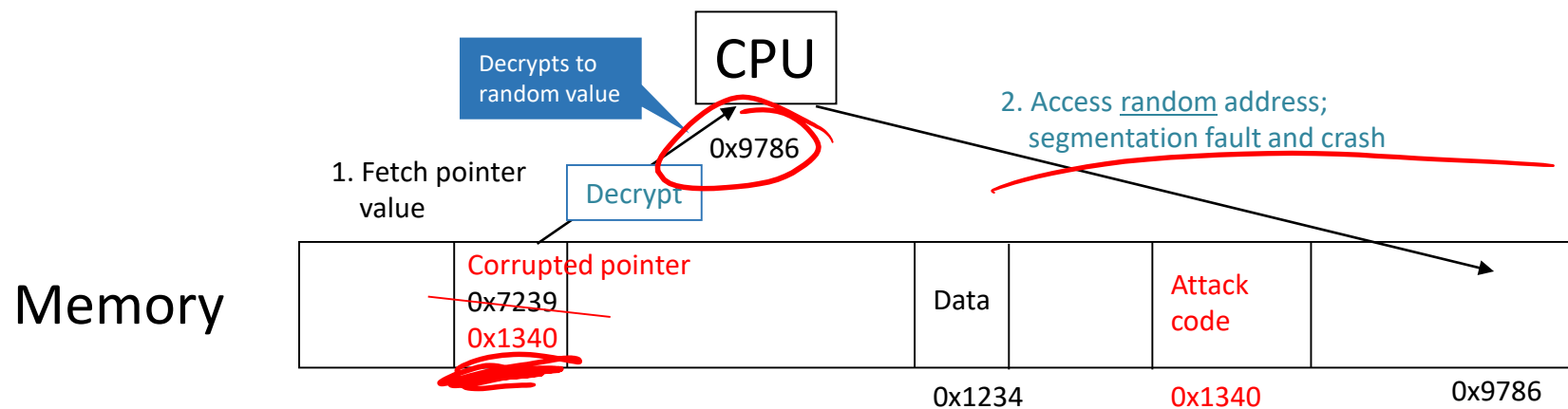
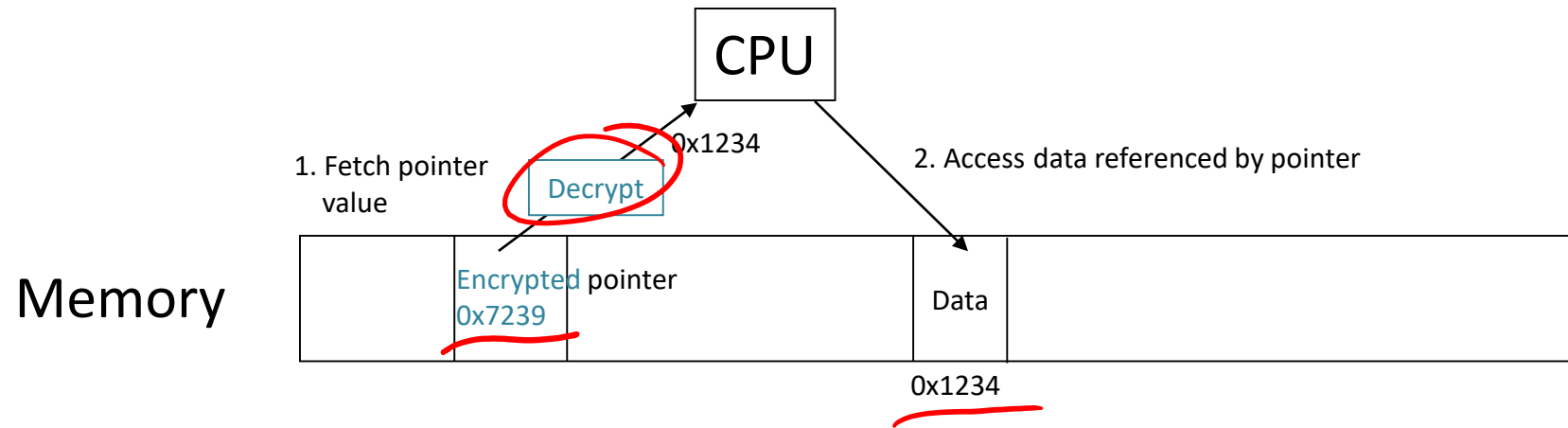
PointGuard

- 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

Normal Pointer Dereference



PointGuard Dereference



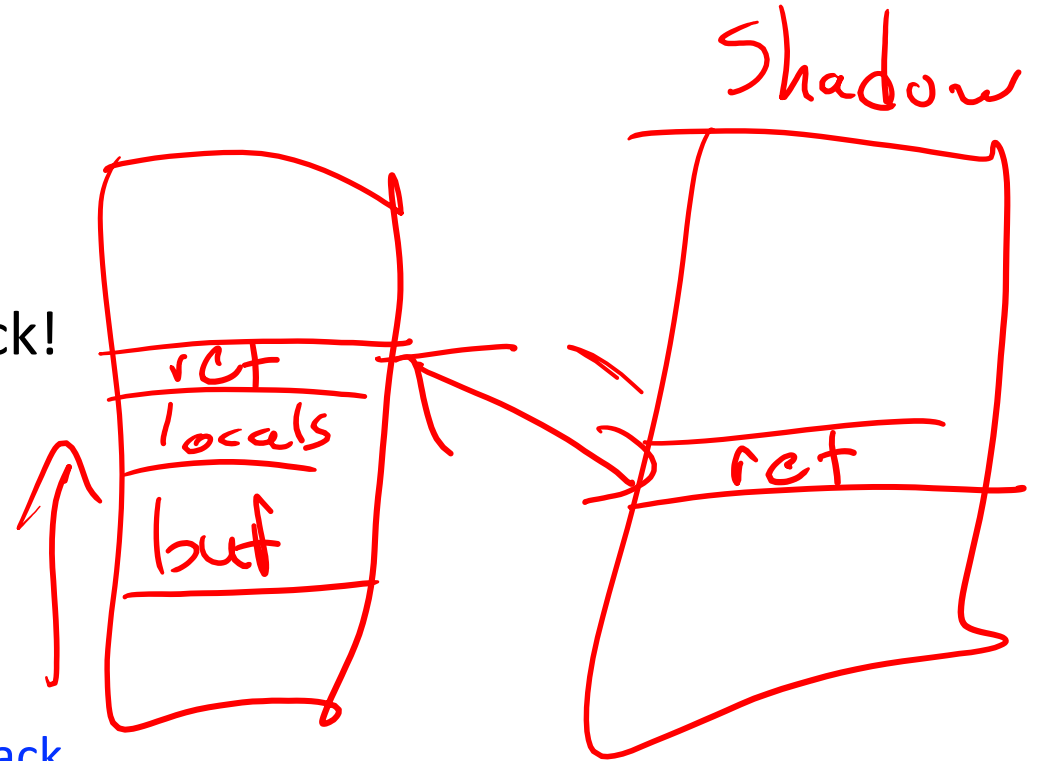
PAC

PointGuard Issues

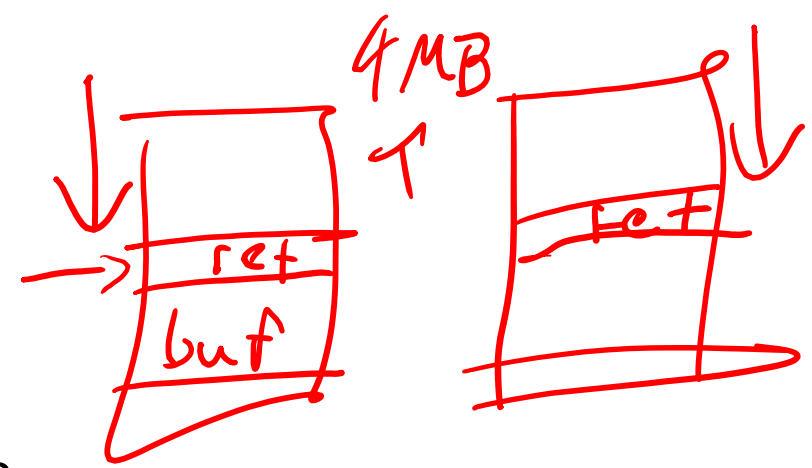
- Must be very fast
 - Pointer dereferences are very common
- Compiler issues
 - Must encrypt and decrypt only pointers
 - If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
 - Store key in its own non-writable 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?

Defense: Shadow stacks

- Idea: don't store return addresses on the stack!
- Store them on... a **different stack!**
 - A hidden stack
- On function call/return
 - **Store/retrieve the return address from shadow stack**
- Or store on both main stack and shadow stack, and compare for equality at function return
- 2020/2021 Hardware Support emerges (e.g., Intel Tiger Lake, AMD Ryzen PRO 5000)



Challenges With Shadow Stacks



- Where do we put the shadow stack?
 - Can the attacker figure out where it is? Can they access it?
- How fast is it to store/retrieve from the shadow stack?
- How *big* is the shadow stack? ←
- Is this compatible with all software?
- (Still need to consider data corruption attacks, even if attacker can't influence control flow.)

8:41

Other Big Classes of Defenses

Carbon

- Use safe programming languages, e.g., Java, Rust
 - 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"

GO

unsafe

Fuzz Testing

- Generate “random” inputs to program
 - Sometimes conforming to input structures (file formats, etc.)
- See if program crashes
 - If crashes, found a bug
 - Bug may be exploitable
- Surprisingly effective
- Now standard part of development lifecycle

What does a modern program do?

(Mostly normal x86_32)

080491f6 <foo>:

```
80491f6: f3 0f 1e fb    endbr32
80491fa: 55             push    %ebp
80491fb: 89 e5          mov     %esp,%ebp
80491fd: 81 ec c0 01 00 00 sub     $0x1c0,%esp
8049203: 8b 45 08        mov     0x8(%ebp),%eax
8049206: 89 85 40 fe ff ff mov     %eax,-0x1c0(%ebp)
804920c: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
8049212: 89 45 fc        mov     %eax,-0x4(%ebp)
8049215: 31 c0          xor     %eax,%eax
8049217: 8b 85 40 fe ff ff mov     -0x1c0(%ebp),%eax
804921d: 83 c0 04        add     $0x4,%eax
8049220: 8b 00          mov     (%eax),%eax
8049222: 50             push    %eax
8049223: 8d 85 44 fe ff ff lea     -0x1bc(%ebp),%eax
8049229: 50             push    %eax
804922a: e8 81 fe ff ff call    80490b0 <strcpy@plt>
804922f: 83 c4 08        add     $0x8,%esp
8049232: 90             nop
8049233: 8b 55 fc        mov     -0x4(%ebp),%edx
8049236: 65 33 15 14 00 00 00 xor     %gs:0x14,%edx
804923d: 74 05          je      8049244 <foo+0x4c>
804923f: e8 4c fe ff ff call    8049090 <__stack_chk_fail@plt>
8049244: c9             leave
8049245: c3             ret
```

(Lab 1 version)

08049196 <foo>:

```
8049196: 55             push    %ebp
8049197: 89 e5          mov     %esp,%ebp
8049199: 81 ec b8 01 00 00 sub     $0x1b8,%esp
804919f: 8b 45 08        mov     0x8(%ebp),%eax
80491a2: 83 c0 04        add     $0x4,%eax
80491a5: 8b 00          mov     (%eax),%eax
80491a7: 50             push    %eax
80491a8: 8d 85 48 fe ff ff lea     -0x1b8(%ebp),%eax
80491ae: 50             push    %eax
80491af: e8 9c fe ff ff call    8049050 <strcpy@plt>
80491b4: 83 c4 08        add     $0x8,%esp
80491b7: 90             nop
80491b8: c9             leave
80491b9: c3             ret
```

Other Common Software Security Issues...

Another Type of Vulnerability: pollev!

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large");
        return;
    }
    memcpy(buf, p, len);
}
```

Snippet 1

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

Snippet 2

```
void *memcpy(void *dst, const void * src, size_t n);

typedef unsigned int size_t;
```

Implicit Cast

- Consider this code:

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

If **len** is negative, may copy huge amounts of input into buf.

```
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

Integer Overflow

ntoh

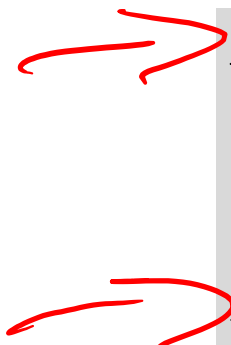
```
size_t len = read_int_from_network();  
char *buf;  
buf = malloc(len+5);  
read(fd, buf, len);
```

- What if **len** is large (e.g., **len = 0xFFFFFFFF**)?
- Then **len + 5 = 4** (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Another Type of Vulnerability

- Consider this code:



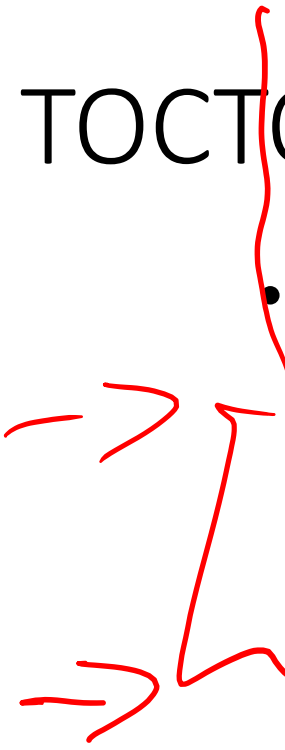
```
if (access("file", W_OK) != 0) {  
    exit(1); // user not allowed to write to file  
}  
  
fd = open("file", O_WRONLY);  
write(fd, buffer, sizeof(buffer));
```

- **Goal:** Write to file only with permission
- What can go wrong?

print ("starting write")

TOCTOU (Race Condition)

- TOCTOU = "Time of Check to Time of Use"



```
if (access("file", W_OK) != 0) {  
    exit(1); // user not allowed to write to file  
}  
  
fd = open("file", O_WRONLY);  
write(fd, buffer, sizeof(buffer));
```

- **Goal:** Write to file only with permission
- Attacker (in another program) can change meaning of "file" between `access` and `open`:
`symlink("/etc/passwd", "file");`

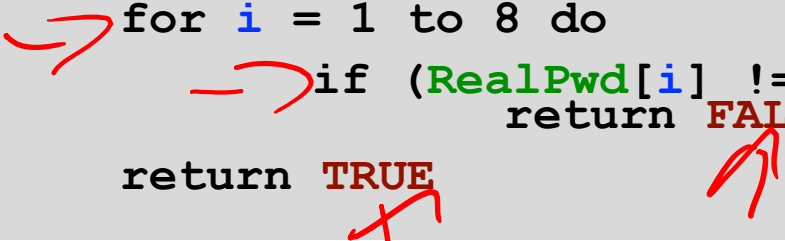
Something Different: Password Checker

- Functional requirements
 - `PwdCheck(RealPwd, CandidatePwd)` should:
 - Return `TRUE` if `RealPwd` matches `CandidatePwd`
 - Return `FALSE` otherwise
 - `RealPwd` and `CandidatePwd` are both 8 characters long

Password Checker

- Functional requirements
 - PwdCheck(RealPwd, CandidatePwd) should:
 - Return TRUE if RealPwd matches CandidatePwd
 - Return FALSE otherwise
 - RealPwd and CandidatePwd are both 8 characters long
- Implementation (like TENEX system)

```
PwdCheck(RealPwd, CandidatePwd) // both 8 chars
  for i = 1 to 8 do
    if (RealPwd[i] != CandidatePwd[i])
      return FALSE
  return TRUE
```



- Clearly meets functional description

TENEX

Attacker Model

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
  for i = 1 to 8 do
    if (RealPwd[i] != CandidatePwd[i])
      return FALSE
  return TRUE
```

- Attacker can guess **CandidatePwds** through some standard interface
- Naive: Try all $256^8 = 18,446,744,073,709,551,616$ possibilities
- Is it possible to derive password more quickly?

aaaaaaa 2ms
baaaaaa 2ms
iaaaaaa 4ms

Try it

dkohlbre.com/cew