UNIVERSITY of SAN FRANCISCO
department of computer science

Buffer overflow exploits

EJ Jung

A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now a computer science professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of fingerd on VAX systems
  - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy
  - Unable to determine remote OS version, worm also attacked fingerd on Suns running BSD, causing them to crash (instead of spawning a new copy)
- For more history:
  - http://www.snowplow.org/tom/worm/worm.html

Buffer Overflow These Days

- Most common cause of Internet attacks
  - Over 50% of advisories published by CERT (computer security incident report team) are caused by various buffer overflows
- Morris worm (1988): overflow in fingerd
  - 6,000 machines infected
- CodeRed (2001): overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in **10 minutes (!!)**

Attacks on Memory Buffers

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
    - If more data is stuffed into it, it spills into adjacent memory
  - If executable code is supplied as "data", victim's machine may be fooled into executing it – we'll see how
    - Code will self-propagate or give attacker control over machine
- First generation exploits: stack smashing
- Second gen: heaps, function pointers, off-by-one
- Third generation: format strings and heap management structures

Stack Buffers

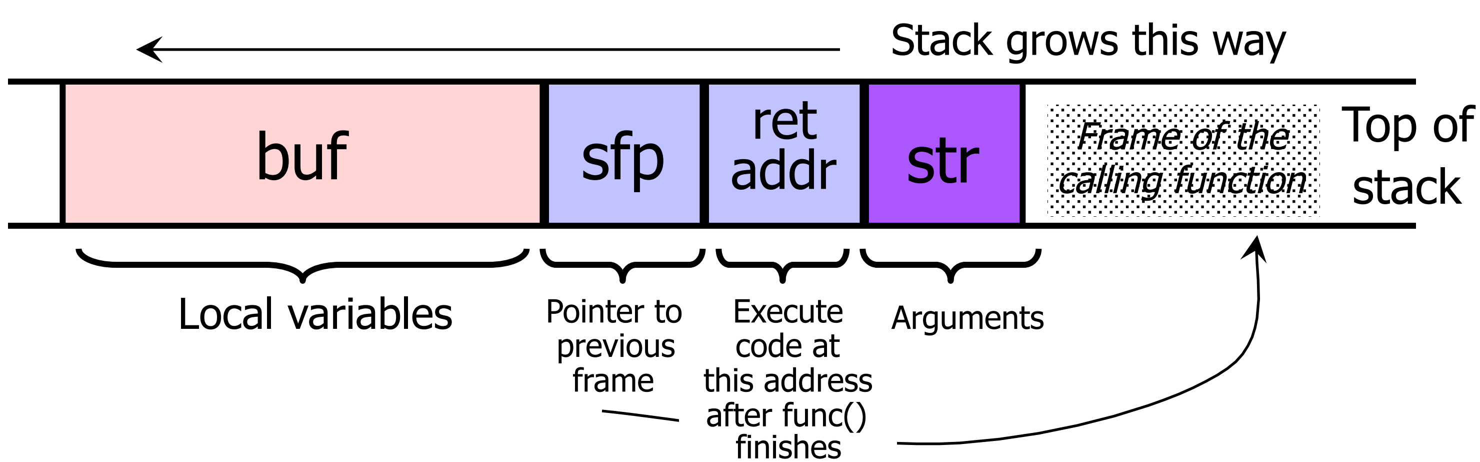
- Suppose Web server contains this function

```
void func(char *str) {
     char buf[126];
     strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack



What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack…

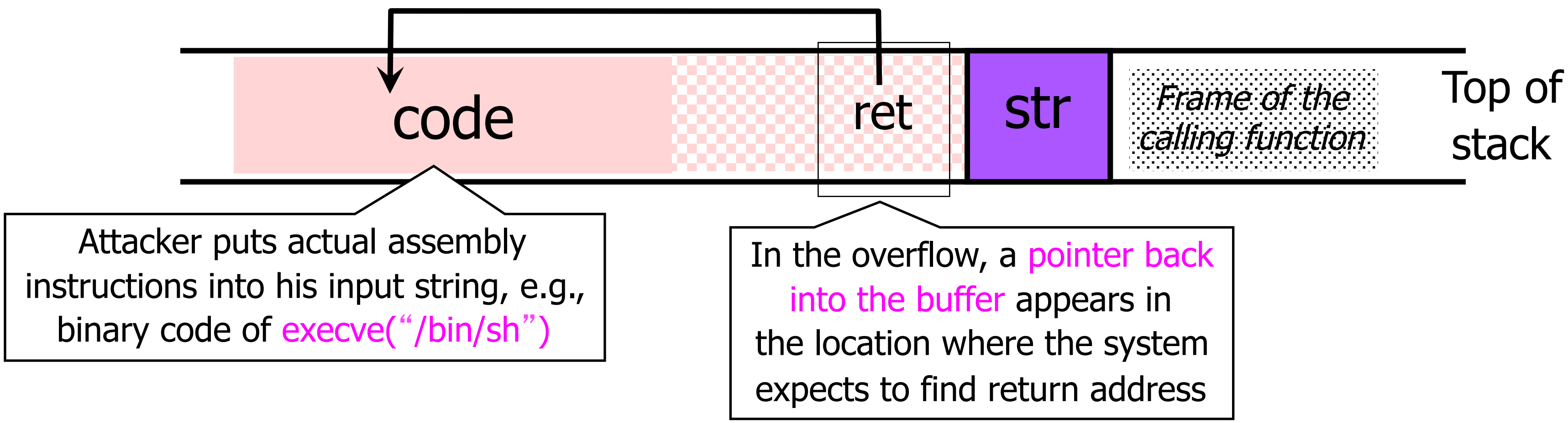
```
void func(char *str) {
      char buf[126];
      strcpy(buf,str);
}
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations

Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

setid

- available on Unix based operating systems
- setuid is a bit indicating temporary privilege associated with the executable
  - similarly setgid
- password change, shell change, …
- insecure executable with setuid is target for buffer overflow

Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Problem: No Range Checking

- strcpy does <u>not</u> check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until "\0" is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, ...)
  - printf(const char *format, ...)

Does Range Checking Help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
    - – Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3):

```
… strcpy(record,user);
  strcat(record,":");
  strcat(record,cpw); …
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

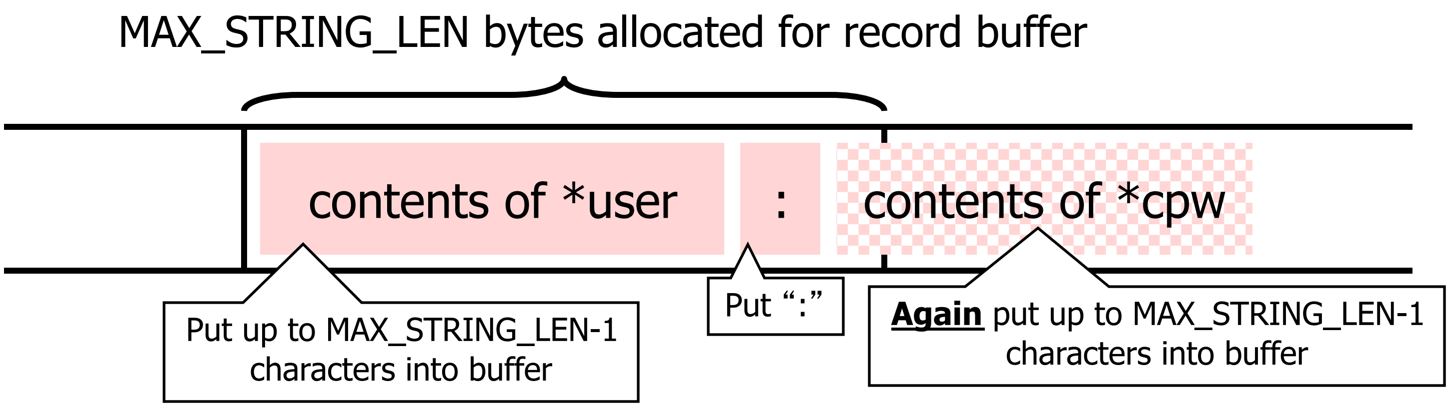
- Published "fix" (do you see the problem?):

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```

Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

```
… strncpy(record,user,MAX_STRING_LEN-1);
  strcat(record,":");
  strncat(record,cpw,MAX_STRING_LEN-1); …
```



Off-By-One Overflow

- Home-brewed range-checking string copy

```
void notSoSafeCopy(char *input) {
      char buffer[512]; int i;

      for (i=0; i<=512; i++)
            buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
      if (argc==2)
            notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

1-byte overflow: can't change RET, but can change pointer to previous stack frame

On little-endian architecture, make it point into buffer

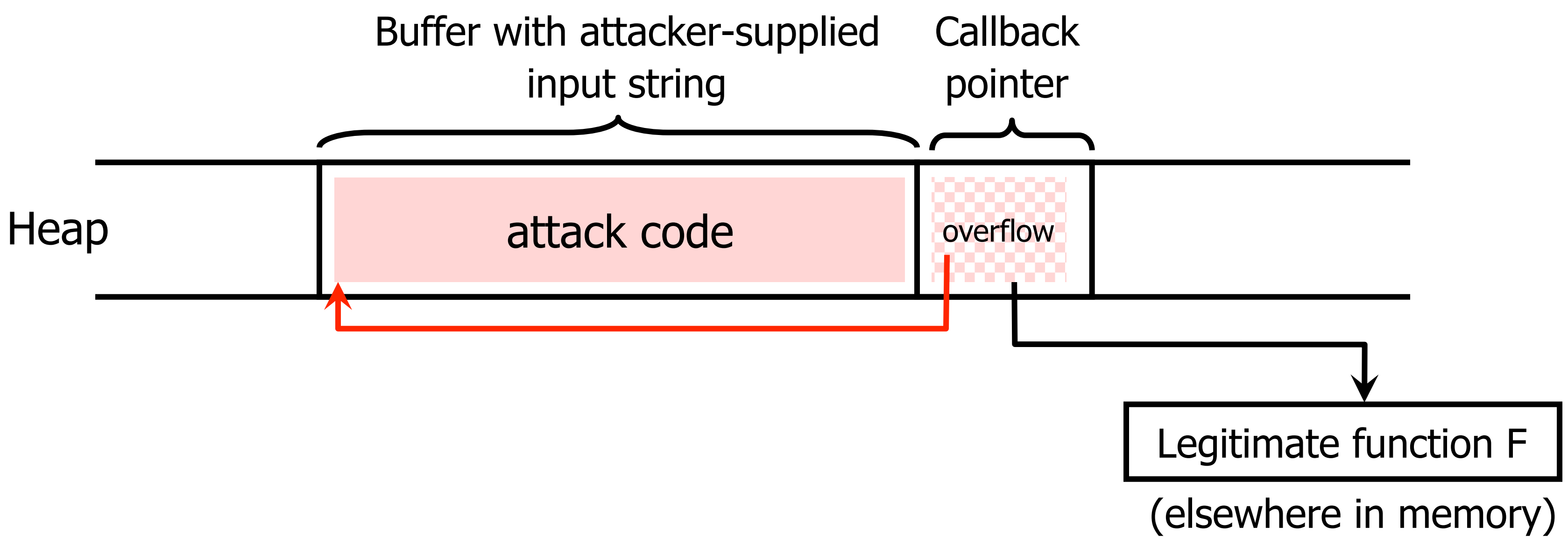
RET for previous function will be read from buffer!

Heap Overflow

- Overflowing buffers on heap can change pointers that point to important data
  - Sometimes can also transfer execution to attack code
  - Can cause program to crash by forcing it to read from an invalid address (segmentation violation)
- Illegitimate privilege elevation: if program with overflow has sysadm/root rights, attacker can use it to write into a normally inaccessible file
  - For example, replace a filename pointer with a pointer into buffer location containing name of a system file
    - Instead of temporary file, write into AUTOEXEC.BAT

Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then another function G can call F as (*P)(…)



Format Strings in C

- Proper use of printf format string:

```
… int foo=1234;
  printf("foo = %d in decimal, %X in hex",foo,foo); …
```

  - This will print

```
foo = 1234 in decimal, 4D2 in hex
```

- Sloppy use of printf format string:

```
… char buf[13]="Hello, world!";
  printf(buf);
  // should've used printf("%s", buf); …
```

  - If buffer contains format symbols starting with %, location pointed to by printf's internal stack pointer will be interpreted as an argument of printf. This can be exploited to move printf's internal stack pointer.

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
… printf("Overflow this!%n",&myVar); …
```

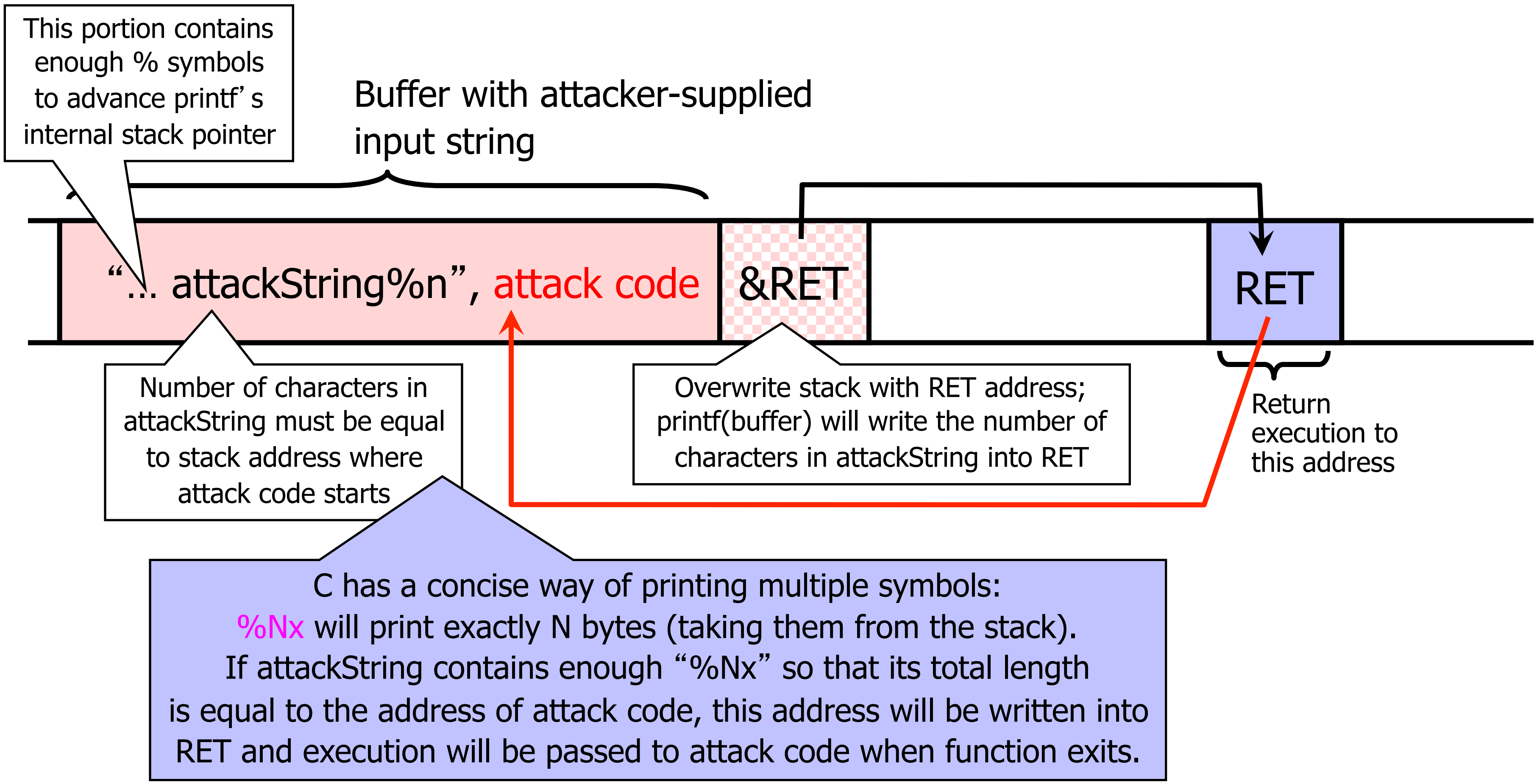
  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- What if printf does not have an argument?

```
… char buf[16]="Overflow this!%n";
  printf(buf); …
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as address into which the number of characters will be written.

Using %n to Mung Return Address



- See "Exploting Format String Vulnerabilities" for details

More Buffer Overflow Targets

- Heap management structures used by malloc()
- URL validation and canonicalization
  - If Web server stores URL in a buffer with overflow, then attacker can gain control by supplying malformed URL
    - Nimda worm propagated itself by utilizing buffer overflow in Microsoft's Internet Information Server
- Some attacks don't even need overflow
  - Naïve security checks may miss URLs that give attacker access to forbidden files
    - For example, http://victim.com/user/../../autoexec.bat may pass naïve check, but give access to system file
    - Defeat checking for "/" in URL by using hex representation

usfCS UNIVERSITY of SAN FRANCISCO department of computer science

Preventing Buffer Overflow

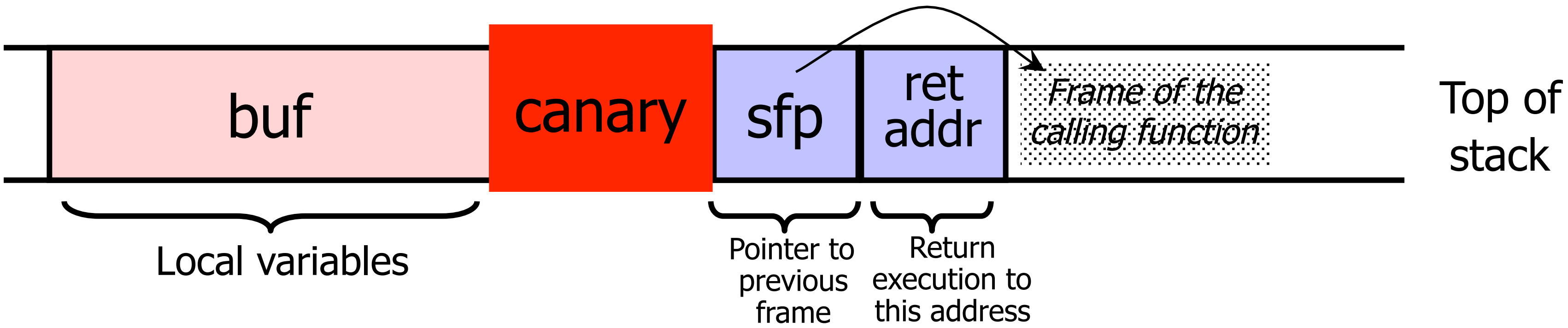
- Use safe programming languages, e.g., Java
  - What about legacy C code?
- Mark stack as non-executable
- Randomize stack location or encrypt return address on stack by XORing with random string
  - Attacker won't know what address to use in his string
- Static analysis of source code to find overflows
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
- Black-box testing with long strings

Non-Executable Stack

- NX bit on every Page Table Entry
  - AMD Athlon 64, Intel P4 “Prescott”, but not 32-bit x86
  - Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD
- Some applications need executable stack
  - For example, LISP interpreters
- Does not defend against return-to-libc exploits
  - Overwrite return address with the address of an existing library function (can still be harmful)
- ...nor against heap and function pointer overflows

Run-Time Checking: StackGuard

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



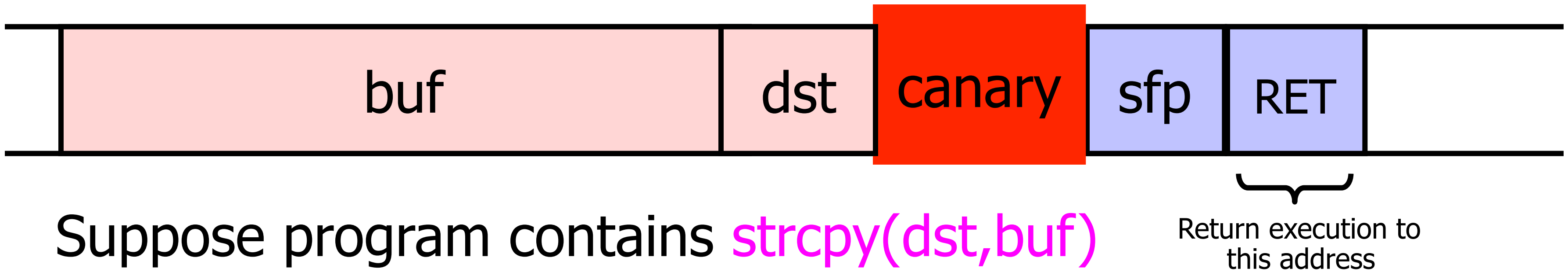
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

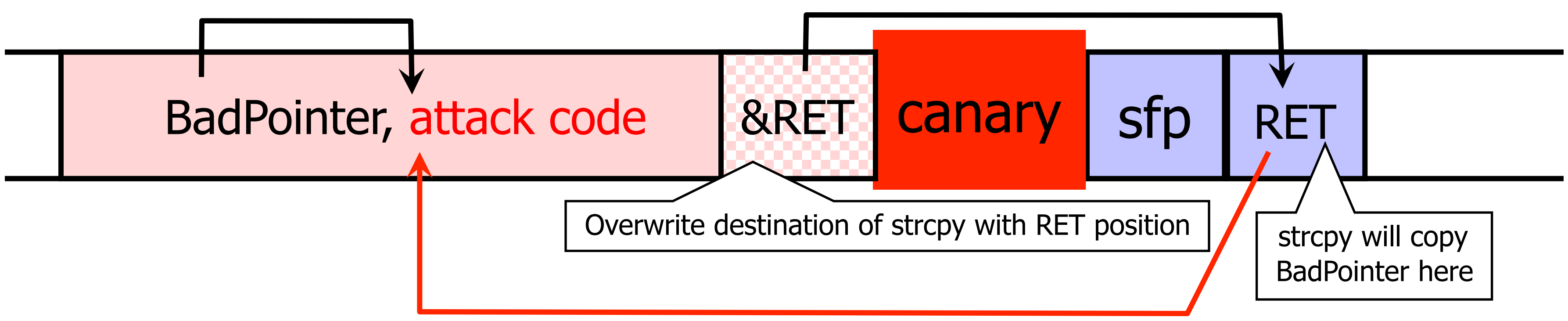
- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- PointGuard also places canaries next to function pointers and setjmp buffers
  - Worse performance penalty
- StackGuard can be defeated!
  - Phrack article by Bulba and Kil3r

Defeating StackGuard (Sketch)

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
  - strcpy will write into RET without touching canary!



Suppose program contains strcpy(dst,buf)

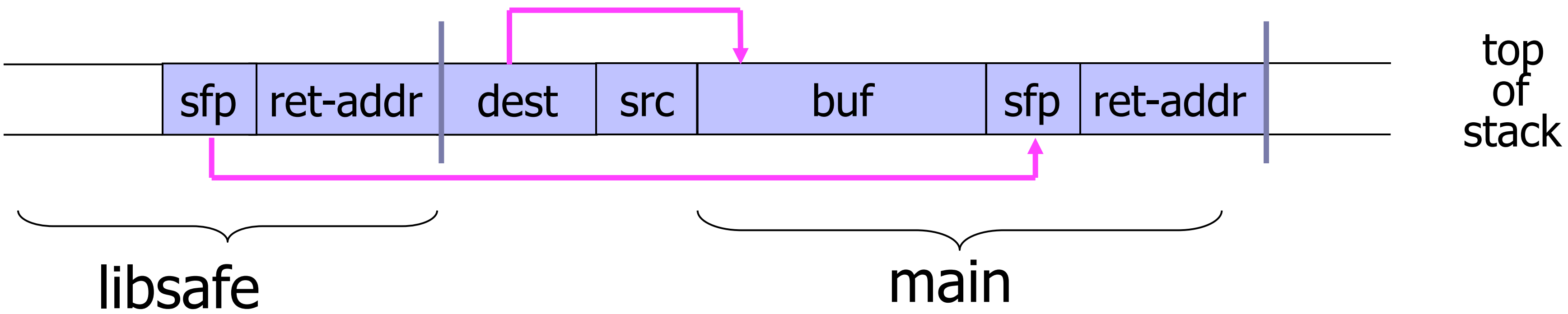


Run-Time Checking: Libsafe

- Dynamically loaded library
- Intercepts calls to strcpy(dest,src)
  - Checks if there is sufficient space in current stack frame

    |frame-pointer – dest| > strlen(src)

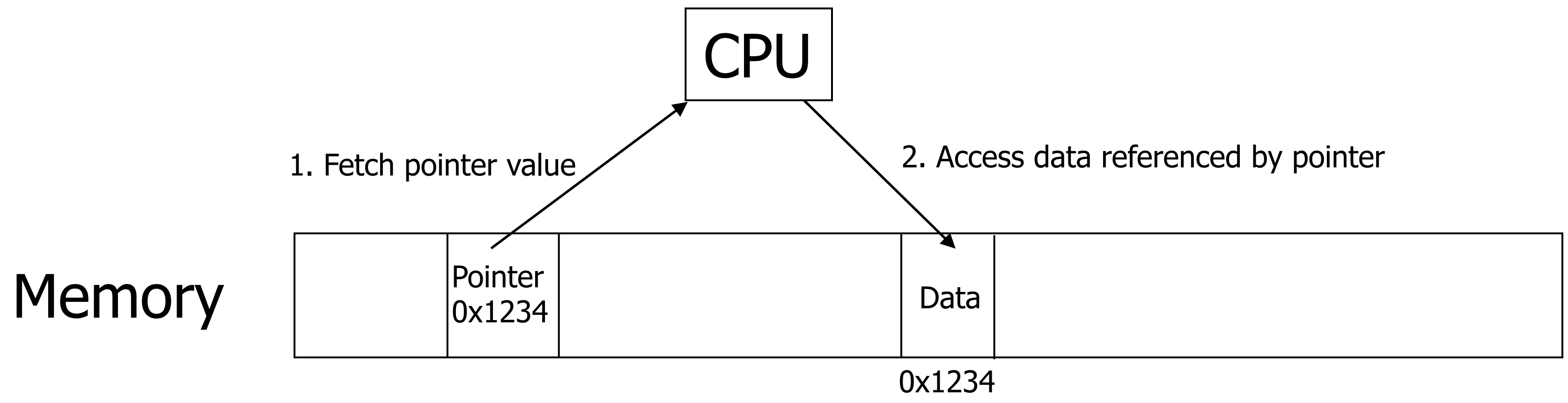
  - If yes, does strcpy; else terminates application

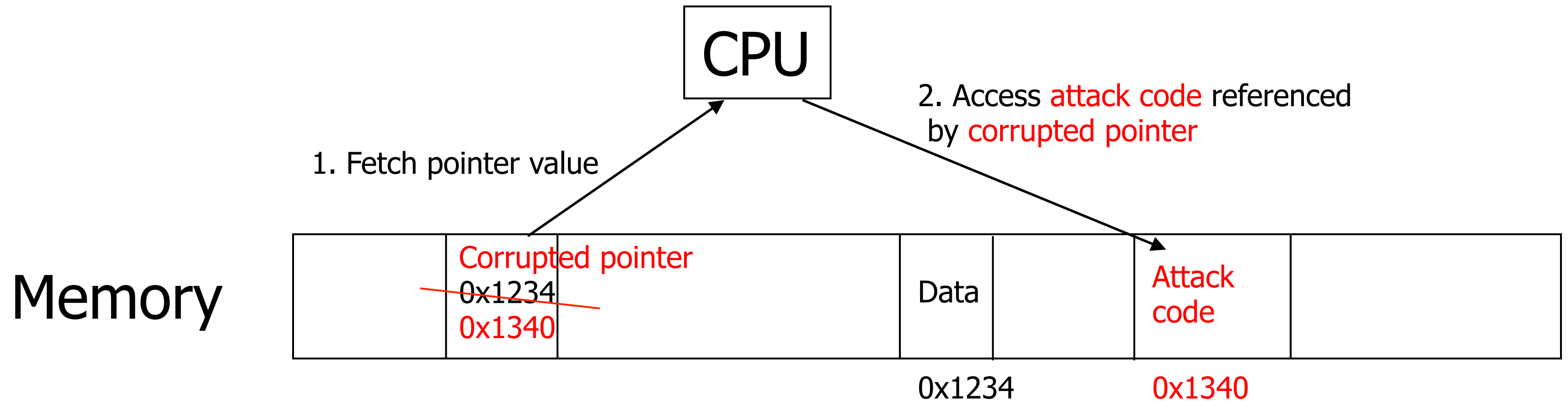


PointGuard

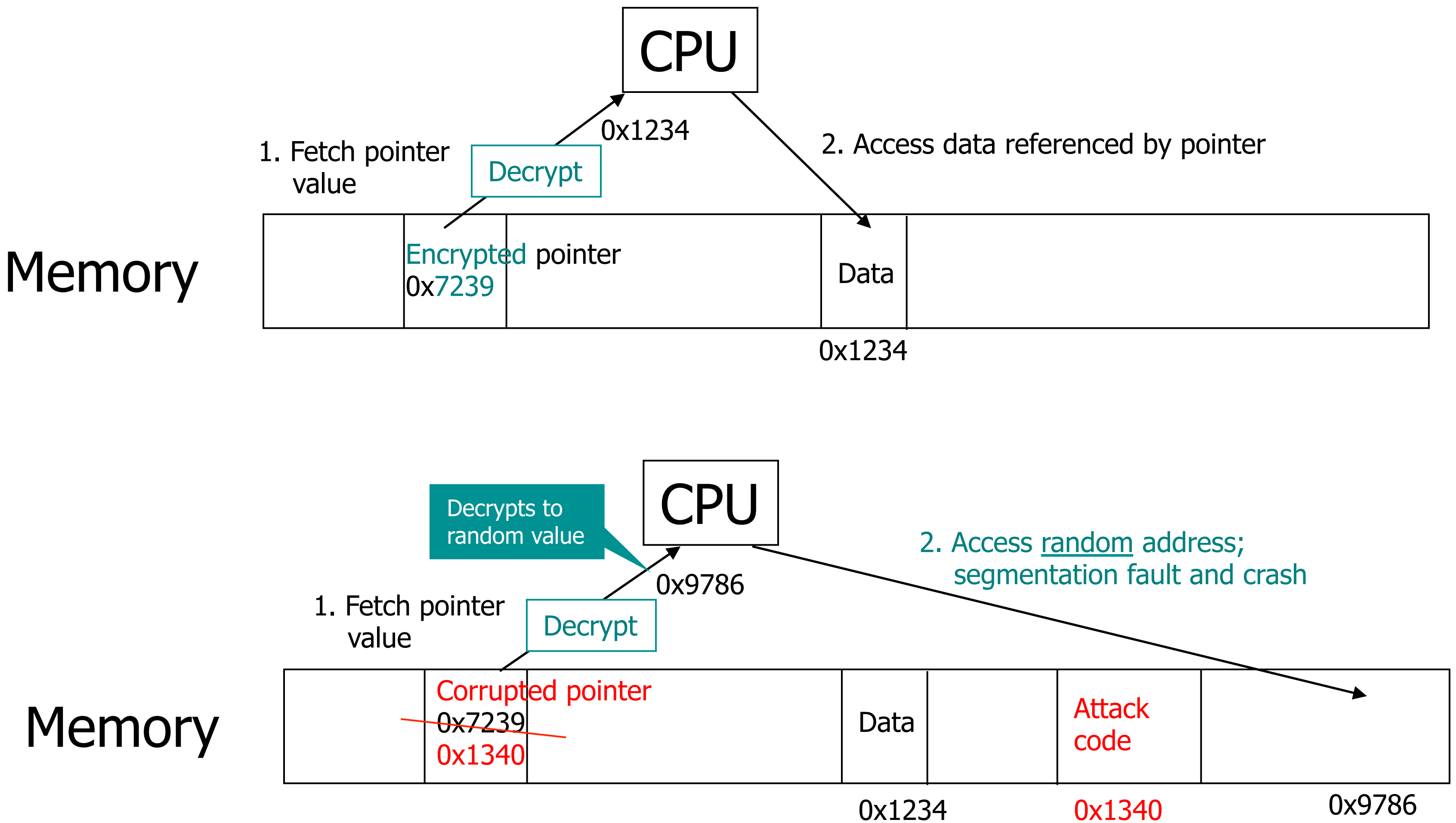
- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflown 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 [Cowan]





PointGuard Dereference [Cowan]



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?