More Exploits of Buffer Overflows and Countermeasures

Yan Huang

Credit: Vitaly Shmatikov (Cornell Tech, CS361)

Review — What’s on the stack?

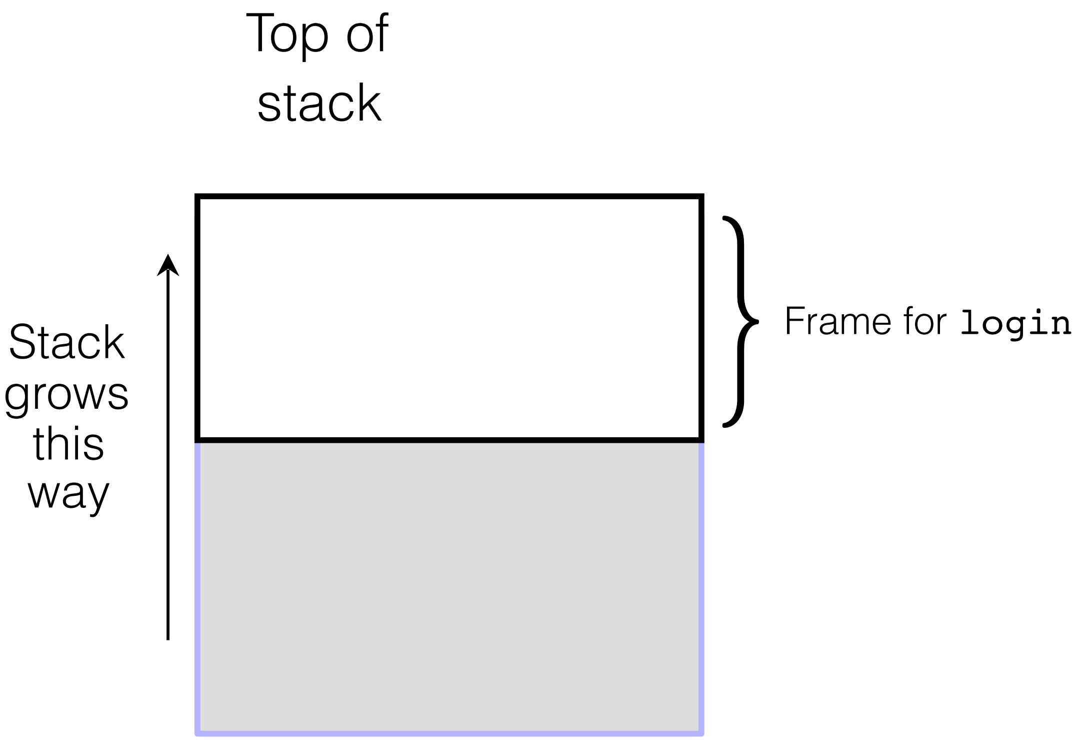
PC →

```
bool num_of_users = 0;

bool login () {
  ……
  if (password_expires())
    reset_password();
  ……
}

void reset_password() {
  ……
  char usr[20], char pwd[100];
  gets(&usr); gets(&pwd);
  update_hash_file(usr,
        compute_hash(pwd, salt));
}
```

Each frame has:
- local data for the function
- a pointer to the previous stack frame (**sfp**)
- the value of previous PC (**ret address**)



Top of stack

Stack grows this way

Frame for `login`

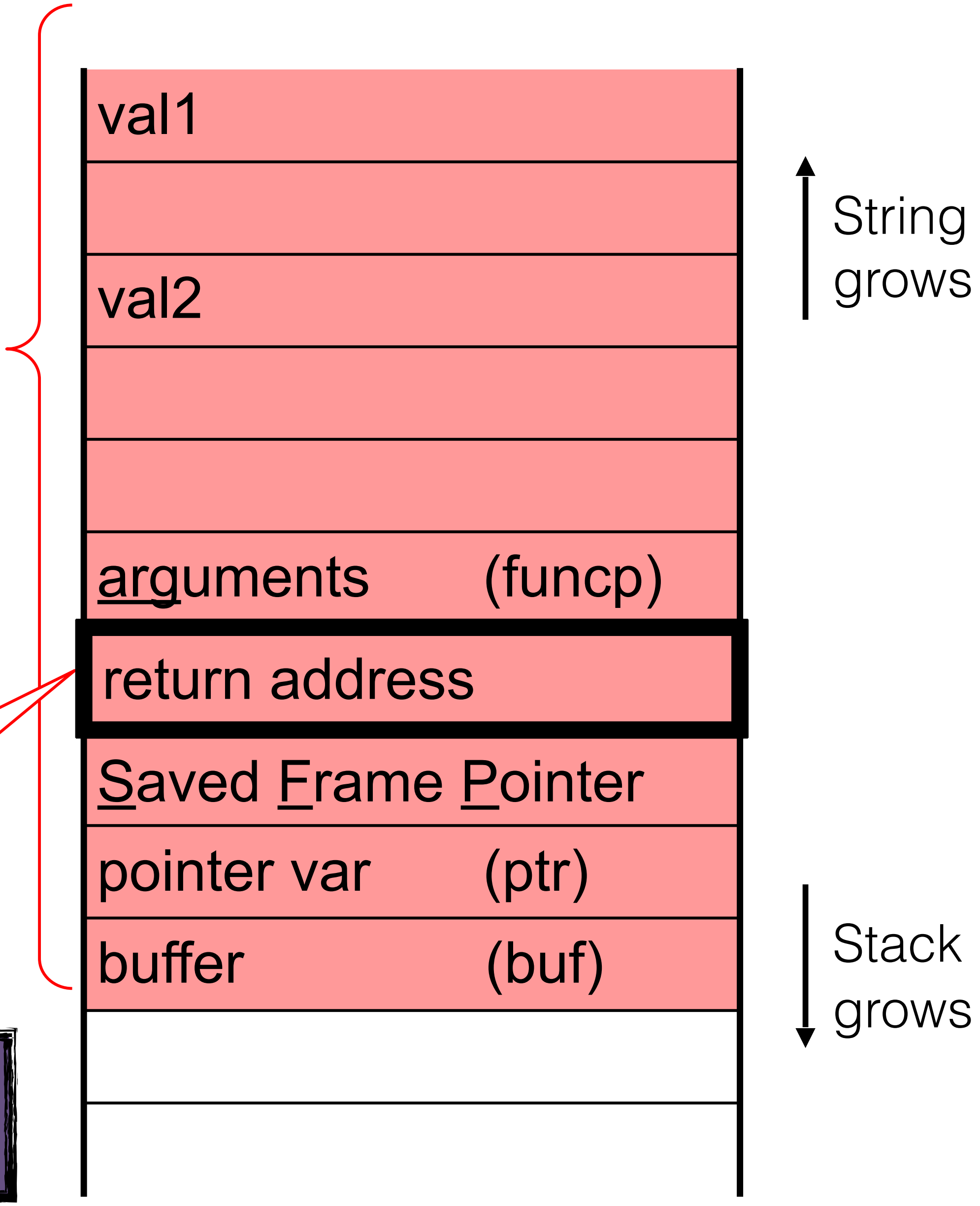
Stack Corruption: General View

```
int bar (int val1) {
    int  val2;
    foo (a_function_pointer);
}
```

Attacker-controlled memory

```
int foo (void (*funcp)()) {
   char* ptr = point_to_an_array;
   char buf[128];
   gets (buf);
   strncpy(ptr, buf, 8);
   (*funcp)();
}
```

Most popular target

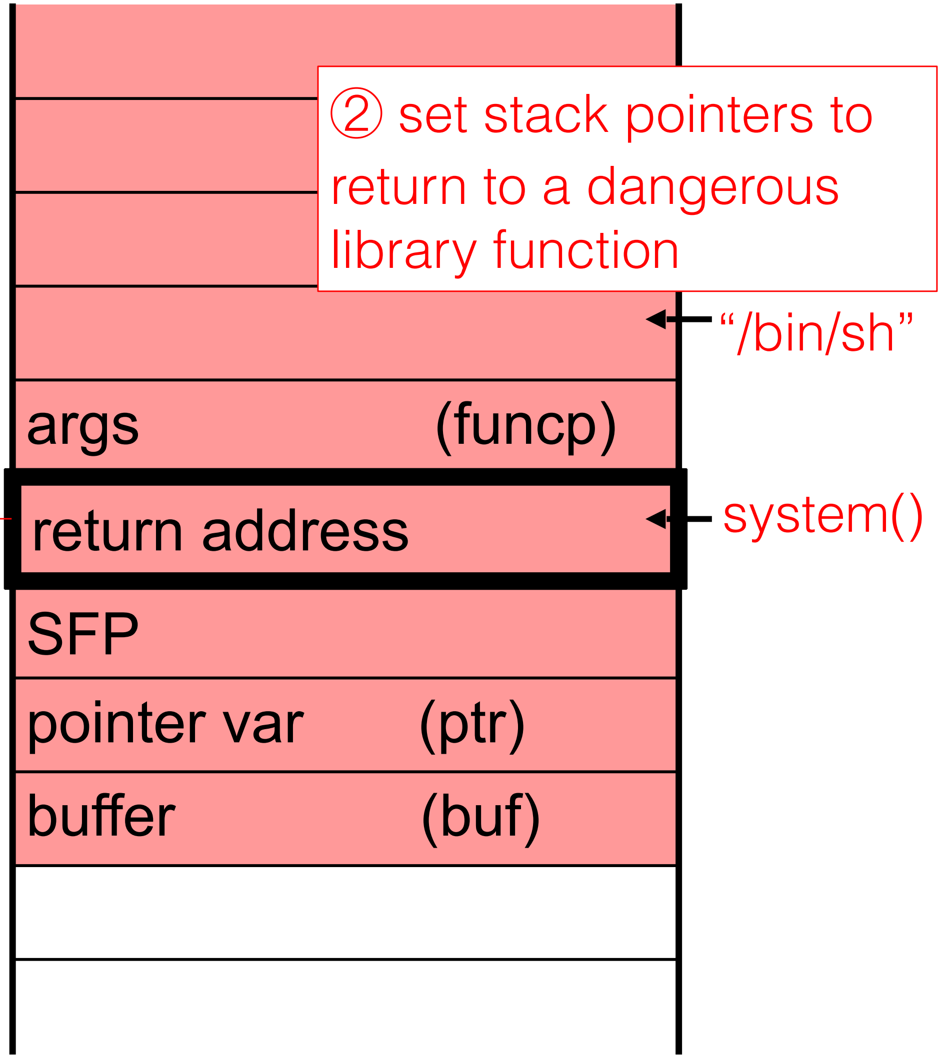


How about dictate the string and stack grow in the same direction?

Attack #1: Return Address

Attack code

①



② set stack pointers to return to a dangerous library function

"/bin/sh"

args (funcp)

return address

system()

SFP

pointer var (ptr)

buffer (buf)

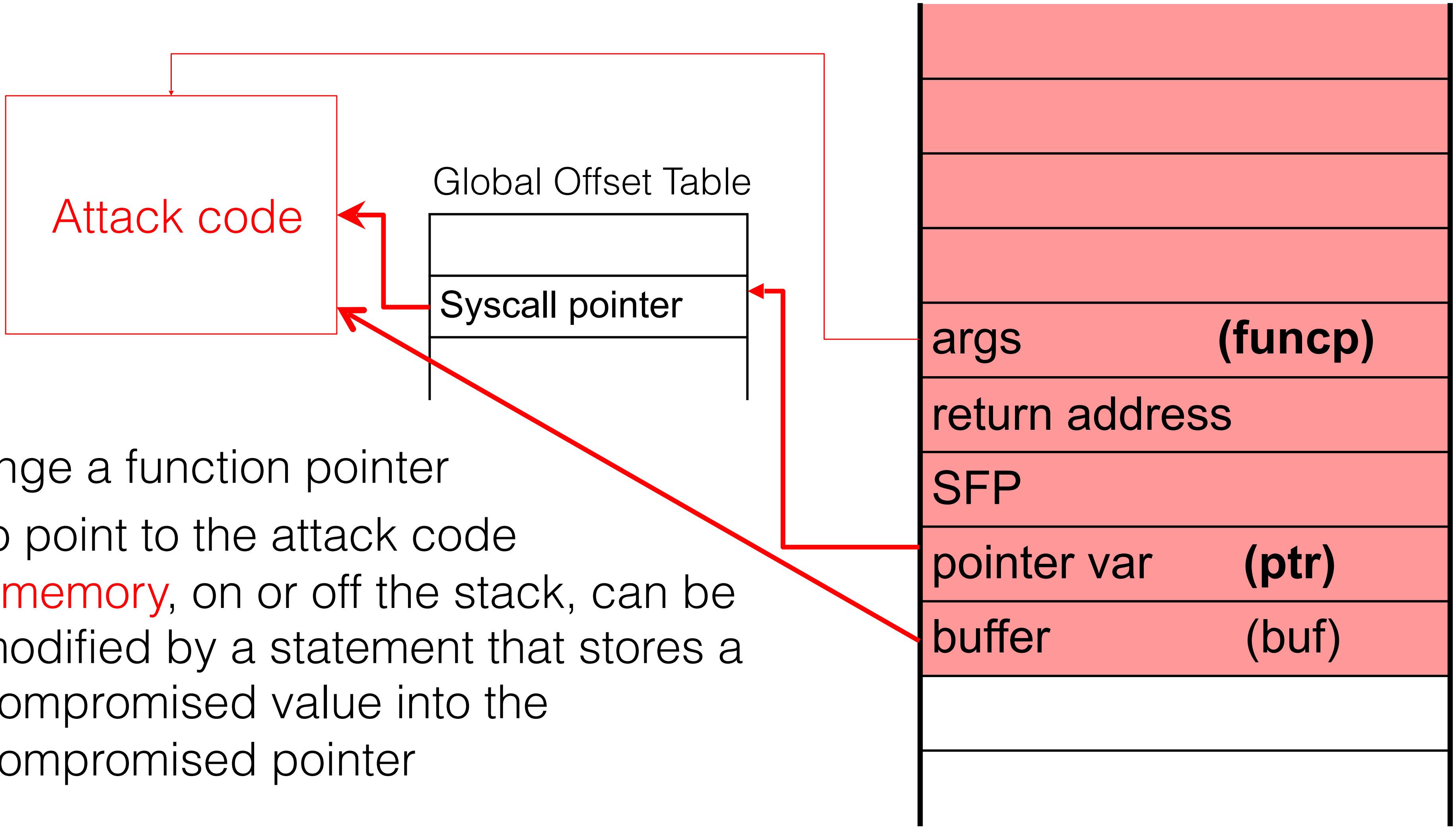
Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

… or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

Basic Stack Code Injection

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- For the basic stack-smashing attack, 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 the "attack assembly code" in the buffer
    - Otherwise application will crash with segmentation violation
  - Attacker must correctly guess the position of his stack buffer when the function is called

Attack #2: Pointer Variables



Change a function pointer to point to the attack code

Any memory, on or off the stack, can be modified by a statement that stores a compromised value into the compromised pointer

```
strcpy(buf, str);
*ptr = buf[0];
```

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]);
}
```

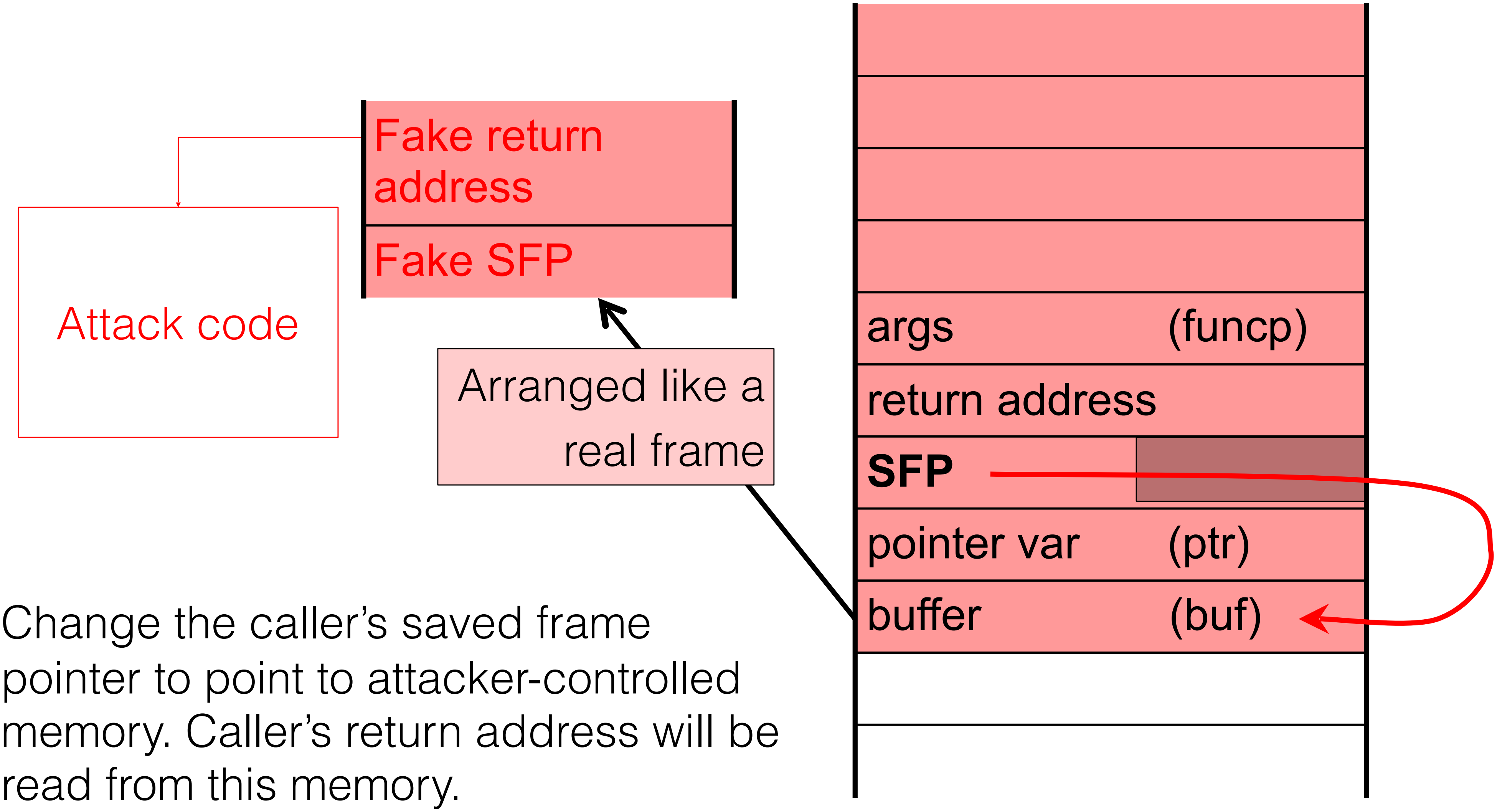
This will copy **<u>513</u>** characters into buffer. Oops!

1-byte overflow: can't change RET, but can change saved pointer to <u>previous</u> stack frame

On little-endian architecture, make it point into buffer

<u>Caller's RET</u> will be read from the buffer!

Attack #3: Frame Pointer



Change the caller’s saved frame pointer to point to attacker-controlled memory. Caller’s return address will be read from this memory.

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
    - Overwrite saved EIP, function callback pointer, etc.
- Idea: prevent execution of untrusted code
  - Make stack and other data areas non-executable
    - Note: messes up useful functionality (e.g., Flash, JavaScript)
  - Digitally sign all code
  - Ensure that all control transfers are into a trusted, approved code image

W⊕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)

Data Execution Prevention - Microsoft Windows

To help protect your computer, Windows has closed this program.

Name: **Windows Explorer**
Publisher: Microsoft Corporation

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

What Does W⊕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⊕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, especially if system() is not available

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! (why?)
  - 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

[Shacham et al.]

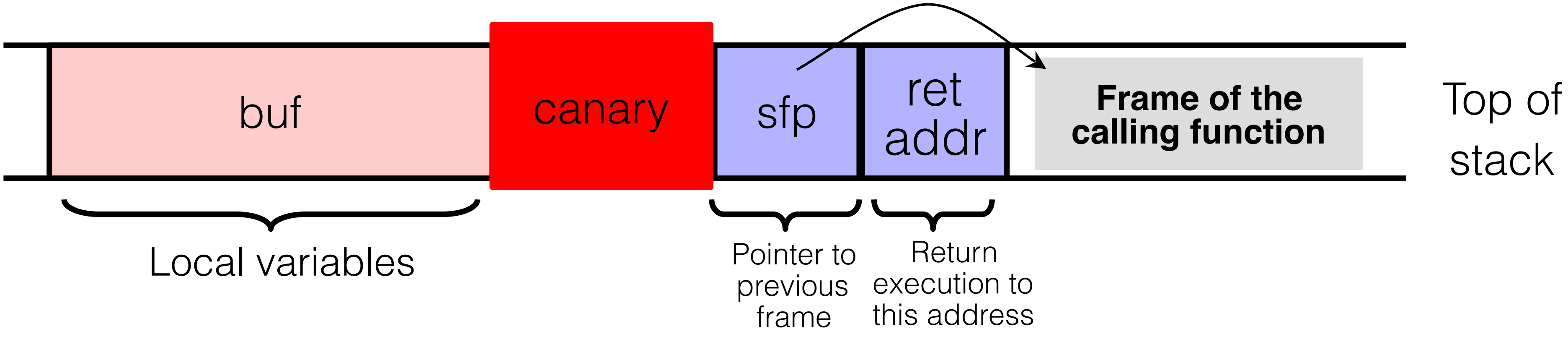
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control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

Other Issues with W⊕X / DEP

- Some applications require executable stack
  - Example: Flash ActionScript, Lisp, other interpreters
- Some applications are not linked with /NXcompat
  - DEP disabled (e.g., some Web browsers)
- JVM makes all its memory RWX – readable, writable, executable (why?)
  - Spray attack code over memory containing Java objects (how?), pass control to them
- “Return” into a memory mapping routine, make page containing attack code writeable

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



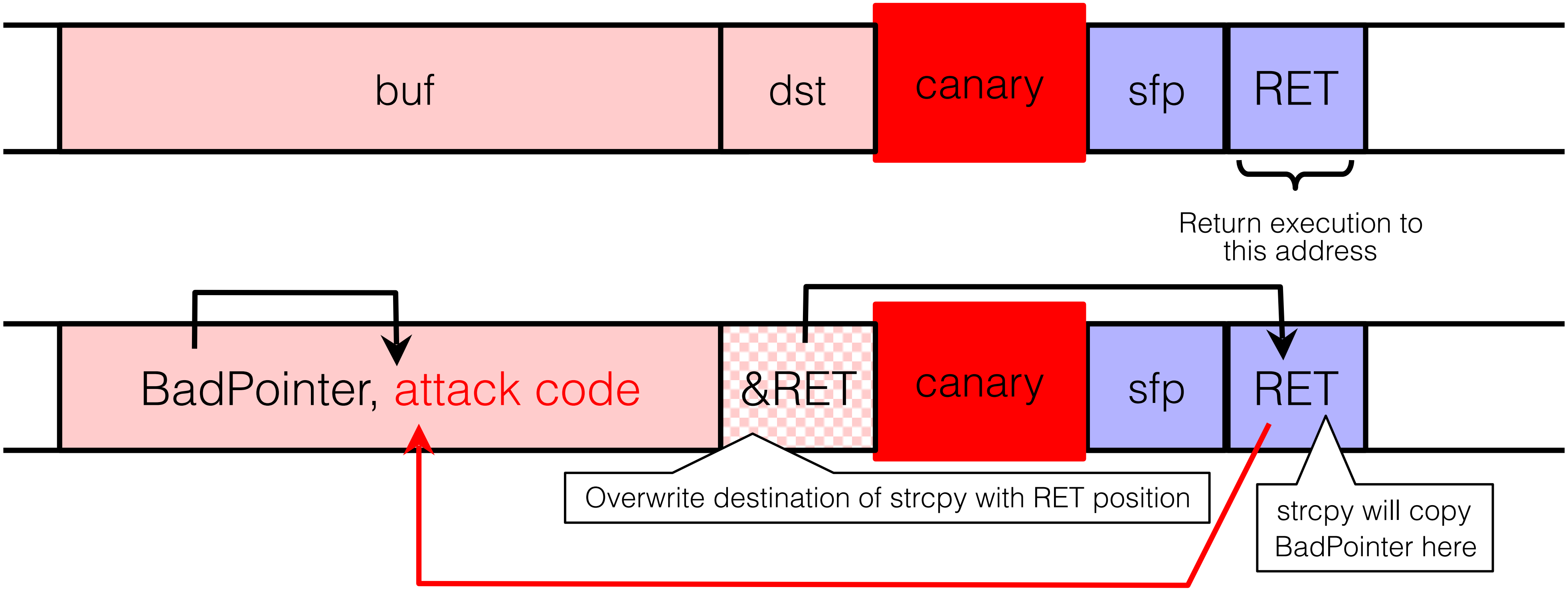
- Candidate Canaries
  - Choose random canary value picked on program start
  - Terminators: “\0”, newline, linefeed, EOF

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



Return execution to this address

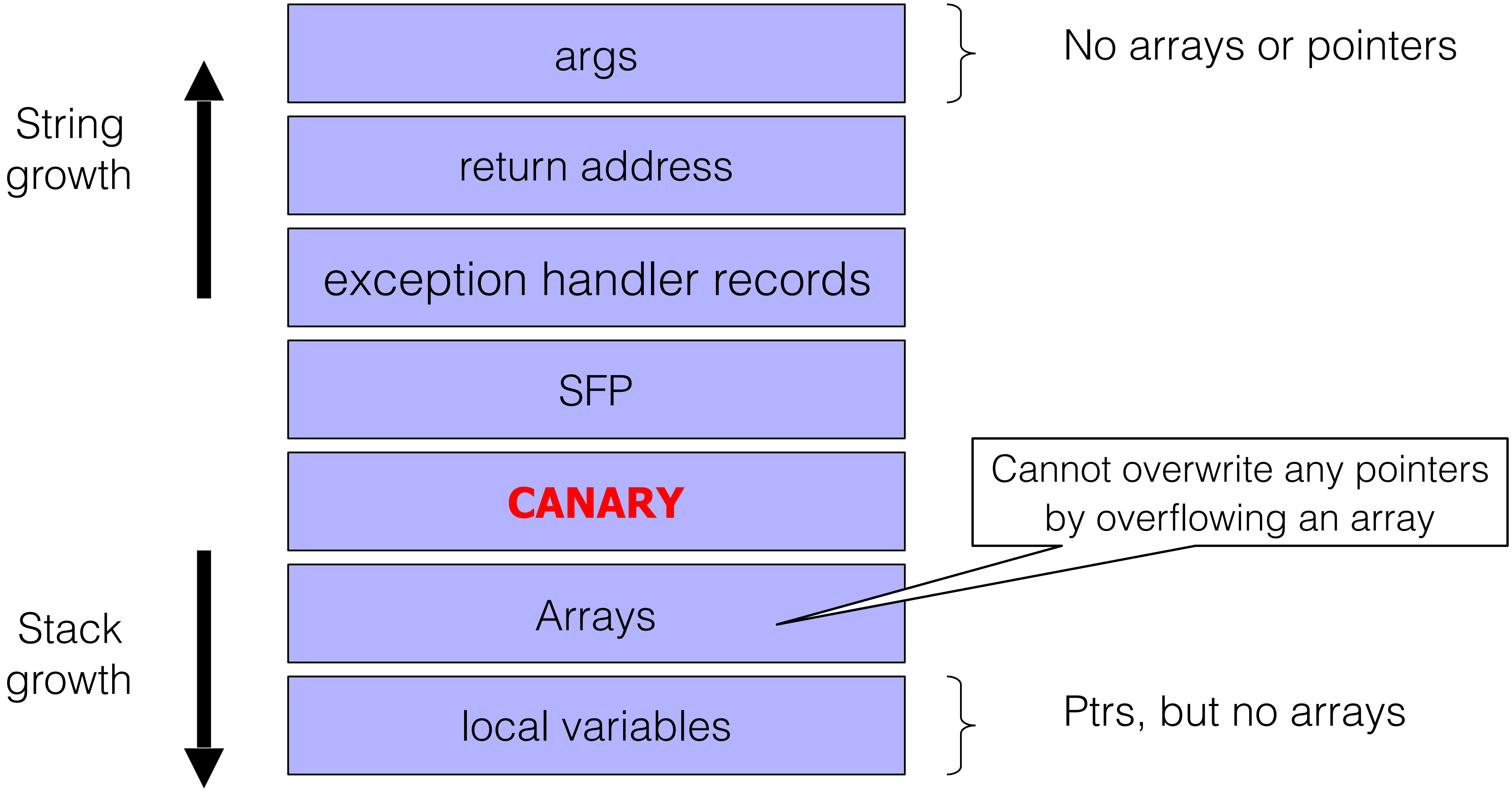
Overwrite destination of strcpy with RET position

strcpy will copy BadPointer here

ProPolice / SSP

[IBM, used in gcc 3.4.1; also MS compilers]

Rerrange stack layout (requires compiler modification)



String growth

Stack growth

args

return address

exception handler records

SFP

CANARY

Arrays

local variables

No arrays or pointers

Cannot overwrite any pointers by overflowing an array

Ptrs, but no arrays

What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Any data stored on the stack
  - Exception handling records
  - Pointers to virtual method tables

    C++: call to a member function passes as an argument “this” pointer to an object on the stack

    Stack overflow can overwrite this object’s vtable pointer and make it point into an attacker-controlled area

    When a virtual function is called (how?), control is transferred to attack code (why?)

    Do canaries help in this case?

    (Hint: when is the integrity of the canary checked?)

Litchfield's Attack

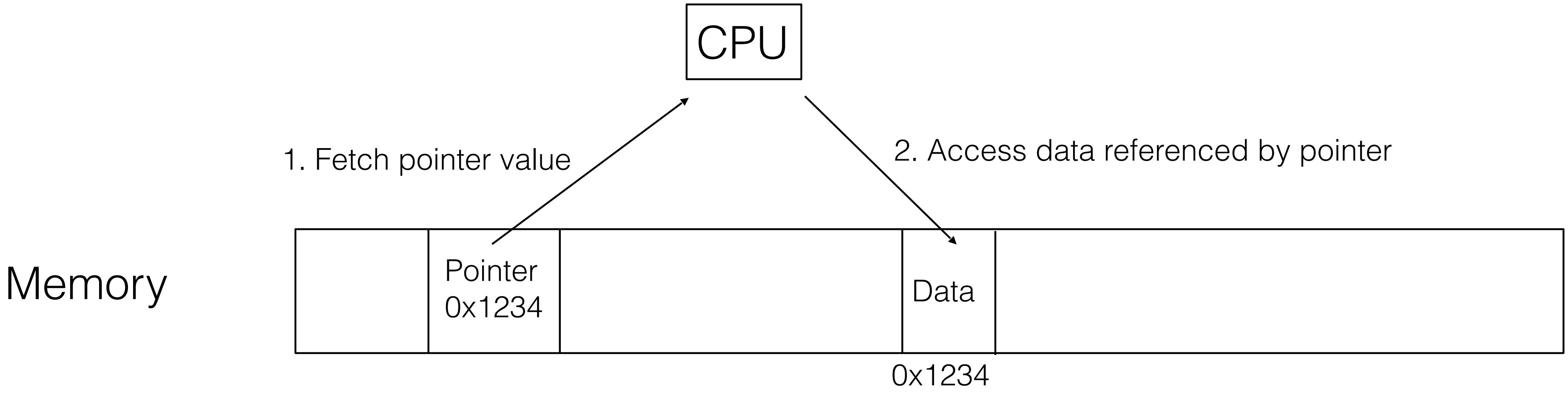
- Microsoft Windows 2003 server implements several defenses against stack overflow
  - Random canary (with /GS option in the .NET compiler)
  - When canary is damaged, exception handler is called
  - Address of exception handler stored on stack above RET
- Attack: smash the canary AND overwrite the pointer to the exception handler with the address of the attack code
  - Attack code must be on heap and outside the module, or else Windows won't execute the fake "handler"
  - Similar exploit used by CodeRed worm

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]



CPU

1. Fetch pointer value

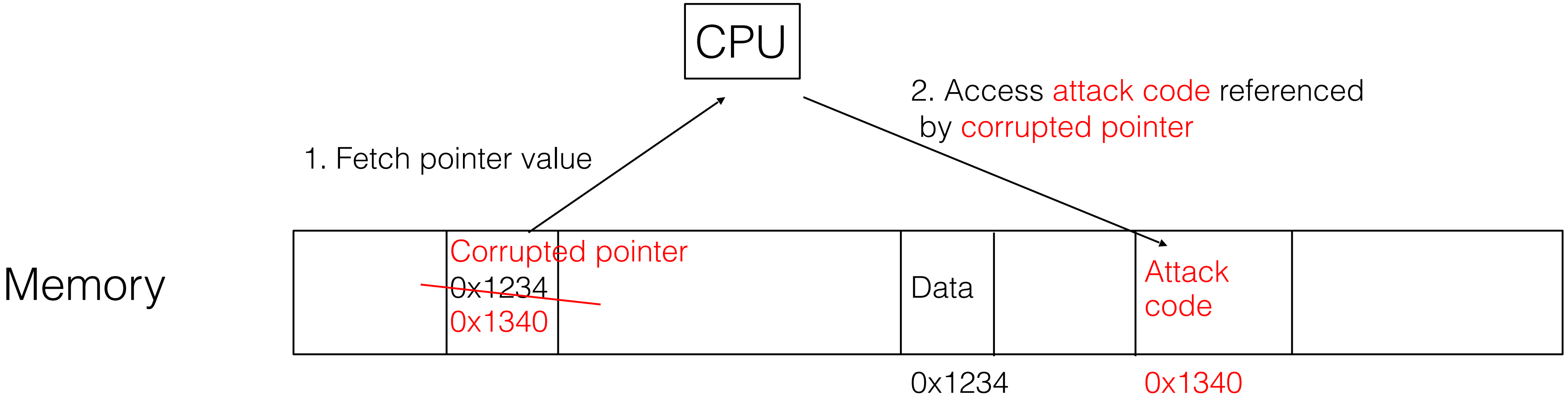
2. Access data referenced by pointer

Memory

Pointer
0x1234

Data

0x1234



CPU

1. Fetch pointer value

2. Access attack code referenced
by corrupted pointer

Memory

Corrupted pointer
~~0x1234~~
0x1340

Data

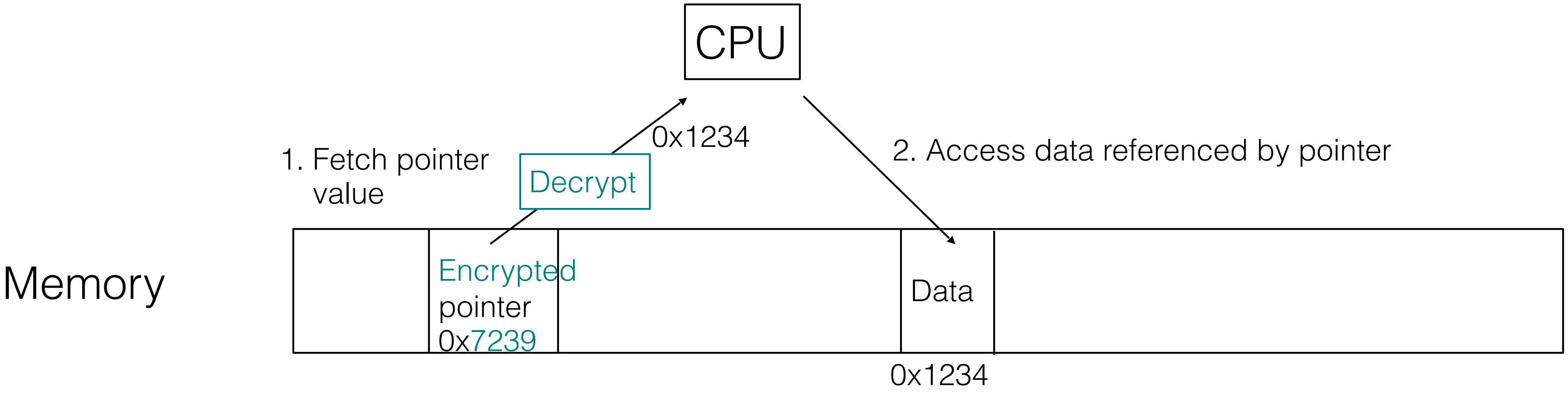
Attack
code

0x1234

0x1340

PointGuard Dereference

[Cowan]



CPU

1. Fetch pointer value

Decrypt

0x1234

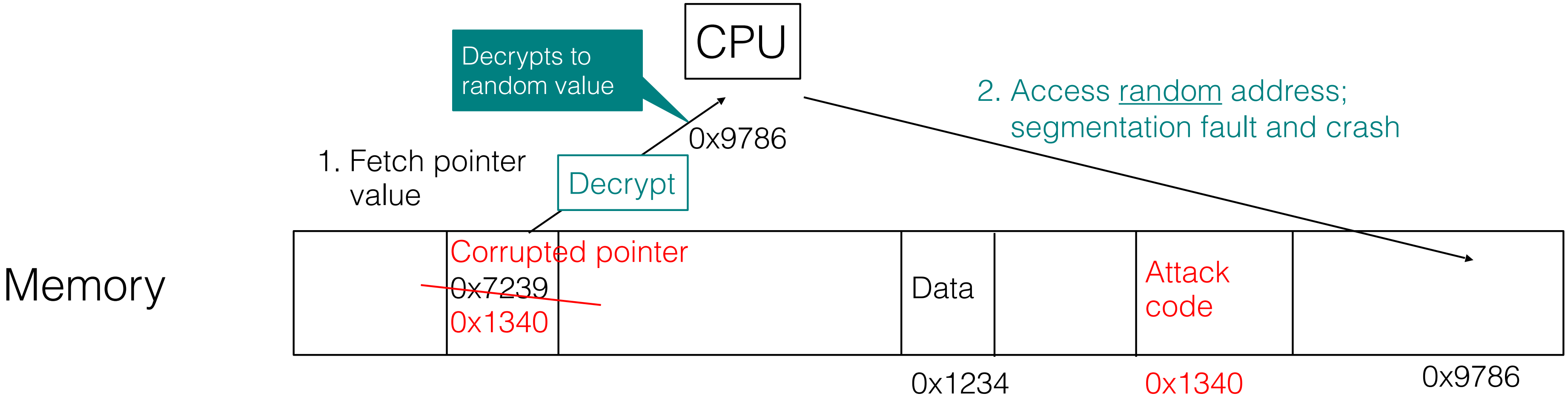
2. Access data referenced by pointer

Memory

Encrypted pointer 0x7239

Data

0x1234



Decrypts to random value

CPU

0x9786

2. Access random address; segmentation fault and crash

1. Fetch pointer value

Decrypt

Memory

Corrupted pointer
~~0x7239~~
0x1340

Data

Attack code

0x1234

0x1340

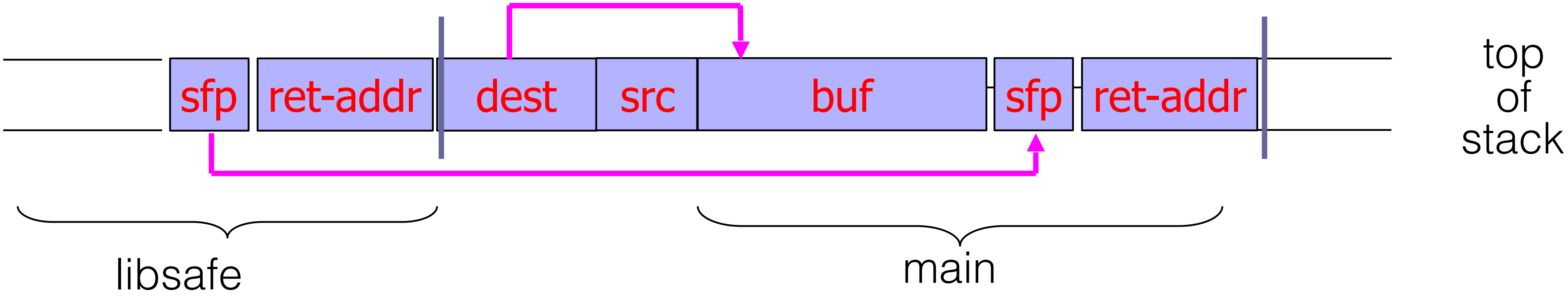
0x9786

PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt <u>only</u> 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 the key in a memory page inaccessible to adversaries
- PG’d code doesn’t mix well with normal code
  - What if PG’d code needs to pass a pointer to OS kernel?

Libsafe

- Dynamically loaded library – no need to recompile!
- Intercepts calls to strcpy(dest, src) and other unsafe C library functions
    - Checks if there is sufficient space in current stack frame |framePointer – dest| > strlen(src)
    - If yes, does strcpy; else terminates application



Limitations of Libsafe

- Protects frame pointer and return address from being overwritten by a stack overflow
- Does not prevent sensitive local variables below the buffer from being overwritten
- Does not prevent overflows on global and dynamically allocated buffers

Charge

- Write a C program with too functions
- Call strcpy to copy a string you supply to an internal buffer on the stack
- Try to subvert your own program with “bad” input
  - a “segmentation fault” counts as a “subversion”