Introduction to Information Security

10. Software Vulnerability

Kihong Heo

KAIST

Software Vulnerability

- A weakness that can be **exploited** by an attacker
  - design flaw, implementation bug, etc
- See CWE (Common Weakness Enumeration) and CVE (Common Vulnerabilities and Exposures)

CWE Definitions

Sort Results By : CWE Number Vulnerability Count

Total number of cwe definitions : 668 Page : 1 (This Page) 2 3 4 5 6 7 8 9 10 11 12 13 14

Select Select&Copy

CWE Number	Name	Number Of Related Vulnerabilities
119	Failure to Constrain Operations within the Bounds of a Memory Buffer	12328
79	Failure to Preserve Web Page Structure ('Cross-site Scripting')	11807
20	Improper Input Validation	7669
200	Information Exposure	6316
89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	5643
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	2968
94	Failure to Control Generation of Code ('Code Injection')	2400
125	Out-of-bounds Read	2122
287	Improper Authentication	1746
284	Access Control (Authorization) Issues	1627
416	Use After Free	1256
190	Integer Overflow or Wraparound	1113
476	NULL Pointer Dereference	900
78	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')	788
787	Out-of-bounds Write	737
362	Race Condition	615
59	Improper Link Resolution Before File Access ('Link Following')	518
77	Improper Sanitization of Special Elements used in a Command ('Command Injection')	489
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	463
611	Information Leak Through XML External Entity File Disclosure	393
434	Unrestricted Upload of File with Dangerous Type	385
732	Incorrect Permission Assignment for Critical Resource	350
74	Failure to Sanitize Data into a Different Plane ('Injection')	327
798	Use of Hard-coded Credentials	319
772	Missing Release of Resource after Effective Lifetime	306
269	Improper Privilege Management	305
601	URL Redirection to Untrusted Site ('Open Redirect')	265
502	Deserialization of Untrusted Data	257
134	Uncontrolled Format String	216
704	Incorrect Type Conversion or Cast	180
415	Double Free	173
522	Insufficiently Protected Credentials	158
532	Information Leak Through Log Files	133
326	Inadequate Encryption Strength	111
369	Divide By Zero	111
285	Improper Access Control (Authorization)	110
306	Missing Authentication for Critical Function	90
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	80
319	Cleartext Transmission of Sensitive Information	76
347	Improper Verification of Cryptographic Signature	76
770	Allocation of Resources Without Limits or Throttling	76
427	Uncontrolled Search Path Element	74
345	Insufficient Verification of Data Authenticity	68
617	Reachable Assertion	68
129	Improper Validation of Array Index	67
327	Use of a Broken or Risky Cryptographic Algorithm	65
668	Exposure of Resource to Wrong Sphere	63
276	Incorrect Default Permissions	60
404	Improper Resource Shutdown or Release	59
311	Missing Encryption of Sensitive Data	56

Heartbleed, 2014
OpenSSL
CVE-2014-0160

goto fail, 2014
MacOS / iOS
CVE-2014-1266

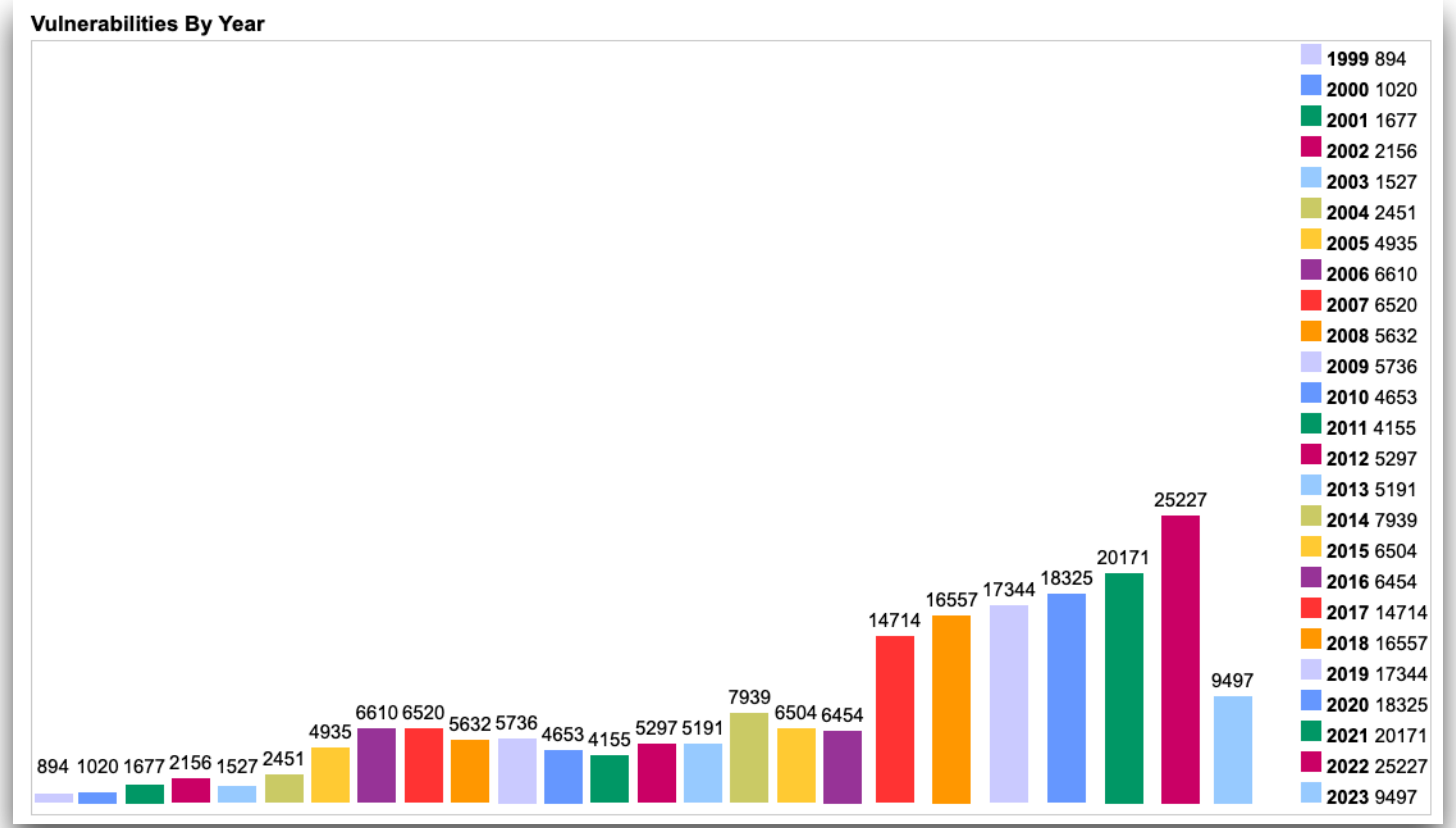
Shellshock, 2014
Bash
CVE-2014-6271

Spectre, 2017
Many CPUs
CVE-2017-5715
CVE-2017-5753

Meltdown, 2017
Many CPUs
CVE-2017-5754

CVEs Over Time

- Gradually increasing over time, why?
- More SW/HW, more bugs, and more powerful analysis tools!



*As of May 2023
https://www.cvedetails.com/browse-by-date.php

Software Security

- Focus on **exploitable** software implementation errors and design flaws
- What happens if someone exploits security vulnerabilities?
  - privilege elevation, arbitrary code execution, access to all files, DoS, etc

SW Bugs as Security Flaws

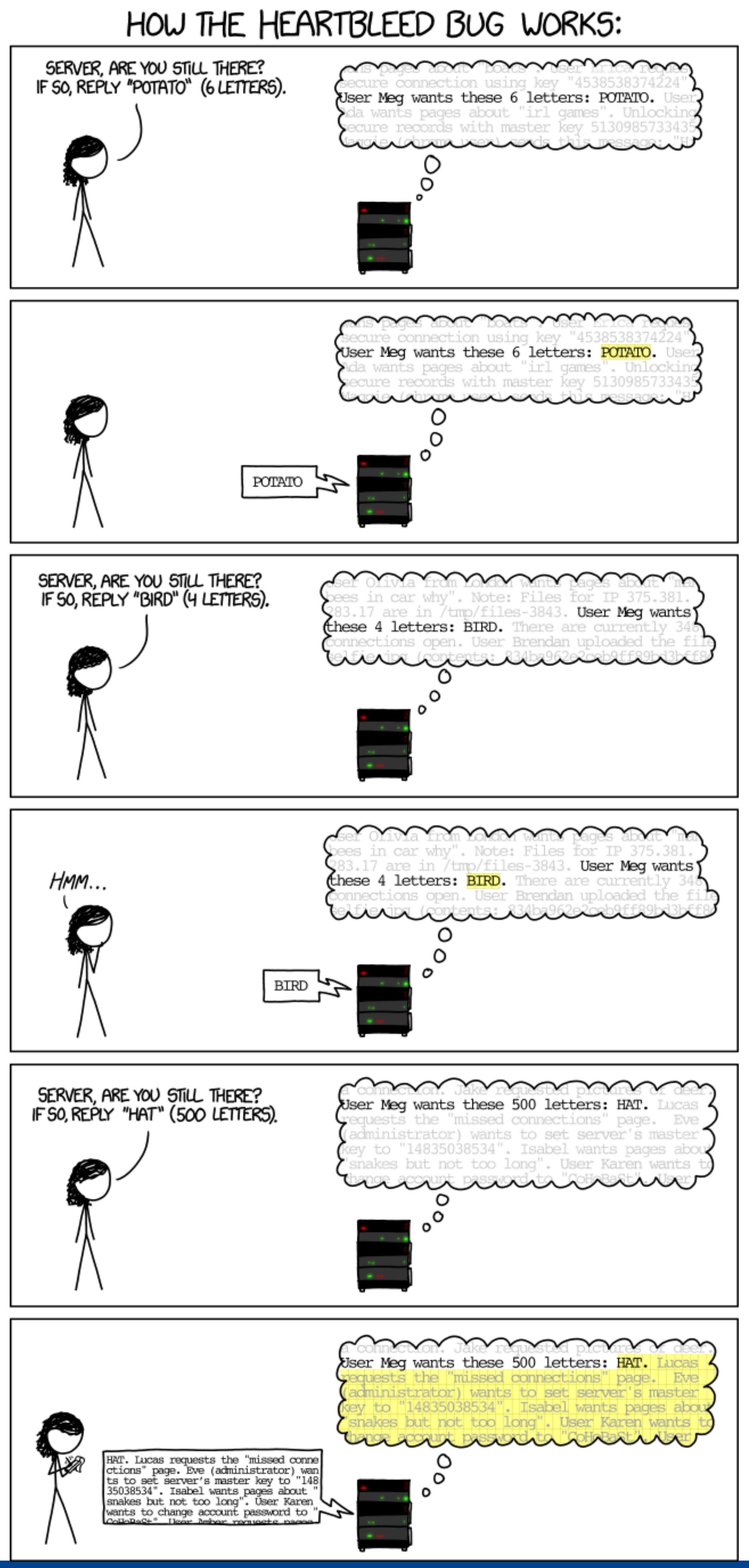
- Main reason of SW security flaws = **SW bug**
- Safety bugs: memory safety violation
  - Usually (but not always) crash the program
  - E.g., buffer-overflow, integer-overflow, division-by-zero, use-after-free, double-free, etc
- Functionality bugs: functional incorrectness
  - E.g., incorrect access control, incorrect SOP, etc

Example: Heartbleed

- A buffer overread bug in OpenSSL
- Attackers can read secret data in memory

Heartbleed, 2014
OpenSSL
CVE-2014-0160

```
// What happens if payload > length of pl?
memcpy(bp, pl, payload);
```



*The comic from https://xkcd.com/1354/

Example: Goto Fail

- A functionality bug in MacOS and iOS
- Attackers can bypass security checks

goto fail, 2014
MacOS / iOS
CVE-2014-1266

```
  ...
  hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
  hashOut.length = SSL_SHA1_DIGEST_LEN;
  if ((err = SSLFreeBuffer(&hashCtx)) != 0)
      goto fail;
  if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;

  err = sslRawVerify(…);
  ...
fail:
  ...
  return err;
```

Example: Goto Fail

- A functionality bug in MacOS and iOS
- Attackers can bypass security checks

goto fail, 2014
MacOS / iOS
CVE-2014-1266

```
  ...
  hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
  hashOut.length = SSL_SHA1_DIGEST_LEN;
  if ((err = SSLFreeBuffer(&hashCtx)) != 0)
      goto fail;
  if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;

  err = sslRawVerify(…);
  ...
fail:
  ...
  return err;
```

Memory Safety

- A property of programs that only use legal memory accesses
- Memory safety violations
  - Spatial: buffer overruns, NULL-dereference, etc
  - Temporal: uninitialized memory read, use-after-free, double-free, etc
- Why are these memory safety violations relevant to security?

Case Study: Buffer Overflows

- Read or write more bytes to a buffer than allocated for it
- Common yet serious problems in languages like C/C++
- Unpredictable outcomes (i.e., undefined behavior)
    - E.g., crash, incorrect output, no effect, etc

```
void myfunction(char *input) {
    int var1, var2;
    char var3[4];
    // what if the length of input > 3?
    strcpy(var3, input);
}
```

Common Memory Layout

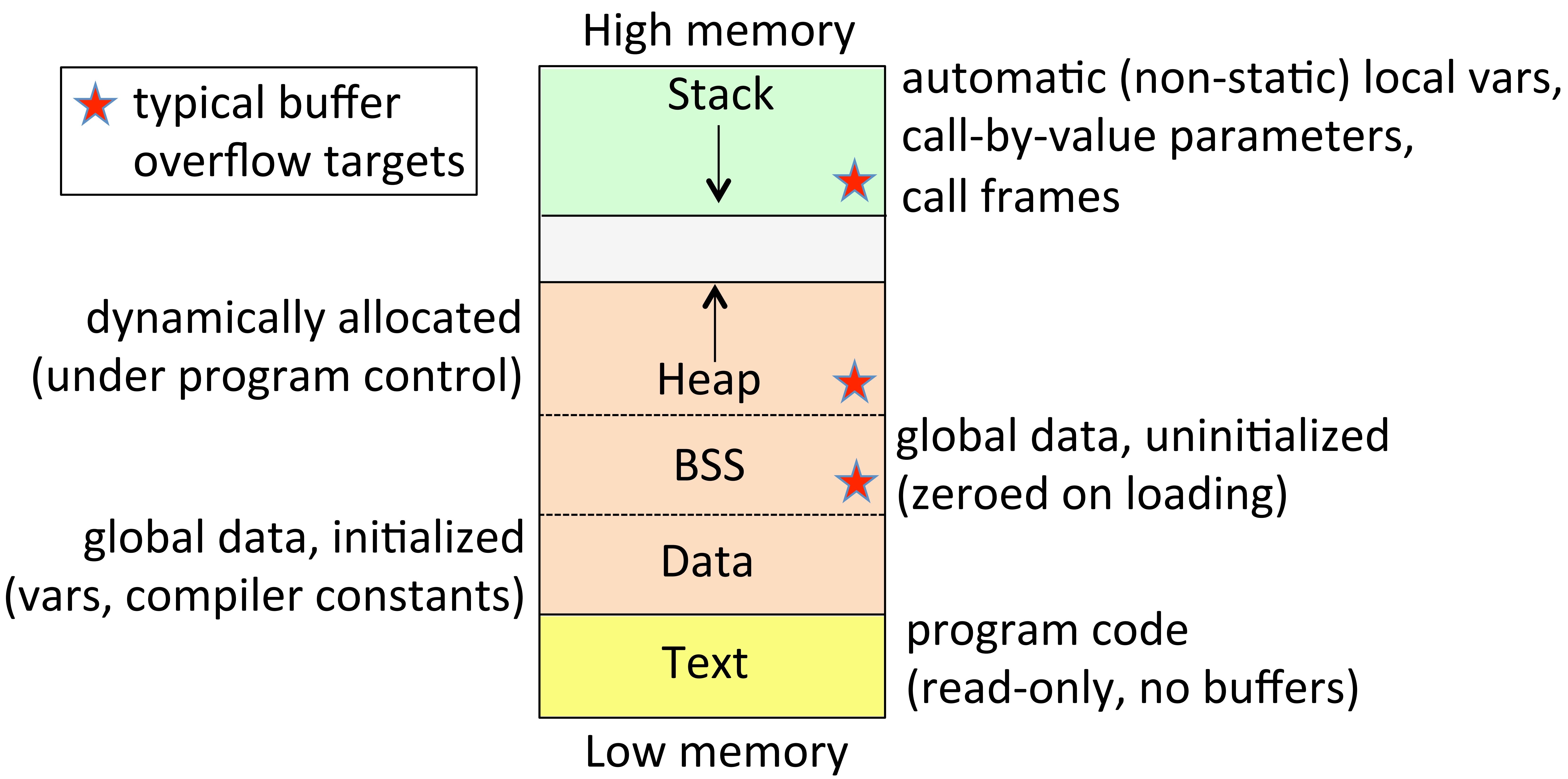
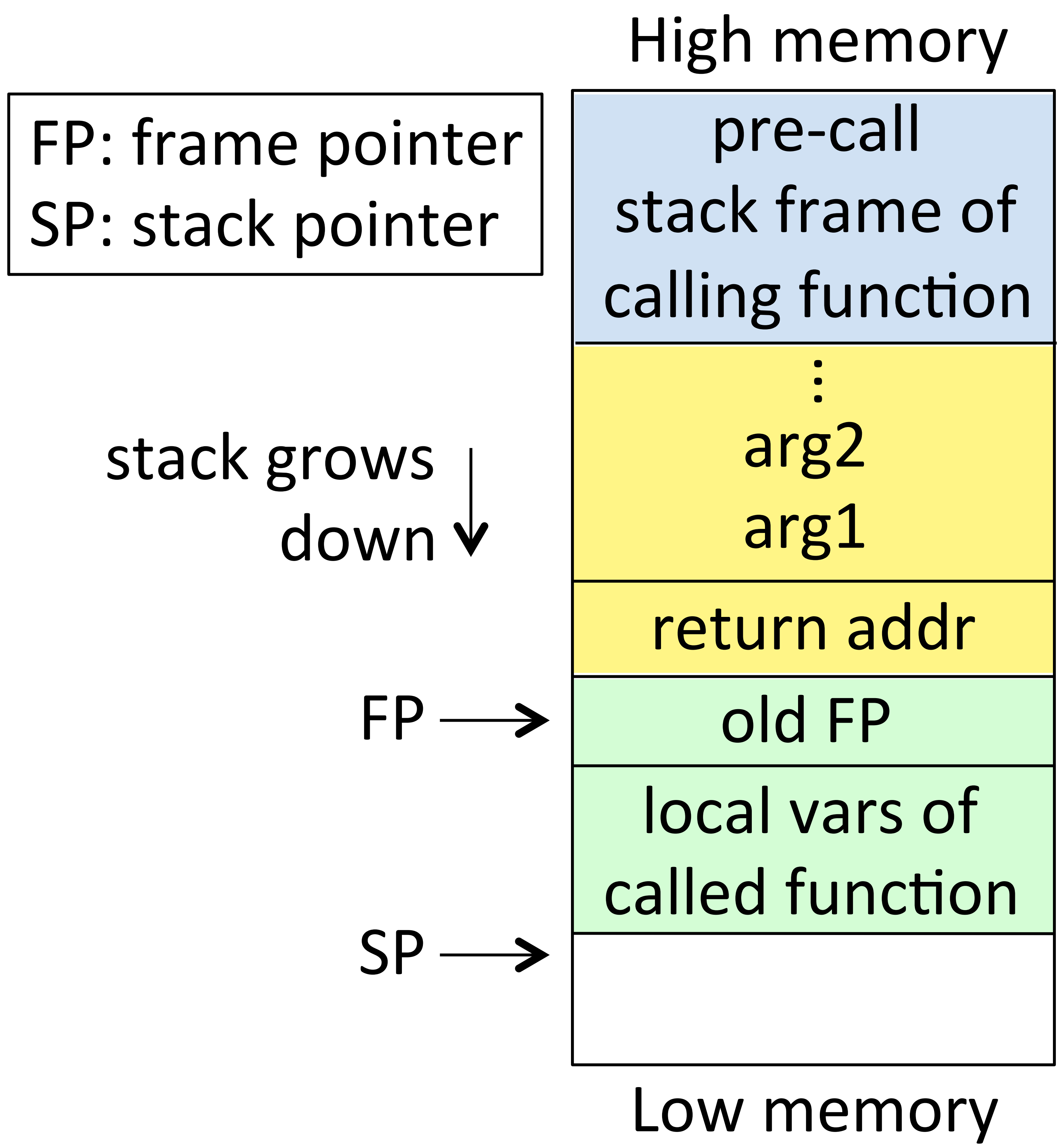


*Figure from Oorschot's book

User-space Stack and Calling Convention



1. calling function pushes args onto stack
2. “call” opcode pushes Instruction Pointer (IP) as return address, then sets IP to begin executing code in called function
3. called function pushes FP for later recovery
4. FP ← SP (so FP points to old FP), now FP+k = args, FP-k = local vars
5. decrement SP, making stack space for local vars
6. called function executes until ready to return
7. called function cleans up stack before return (SP ← FP , FP ← old FP popped from stack)
8. “ret” opcode pops return address into IP, to resume execution back to calling function

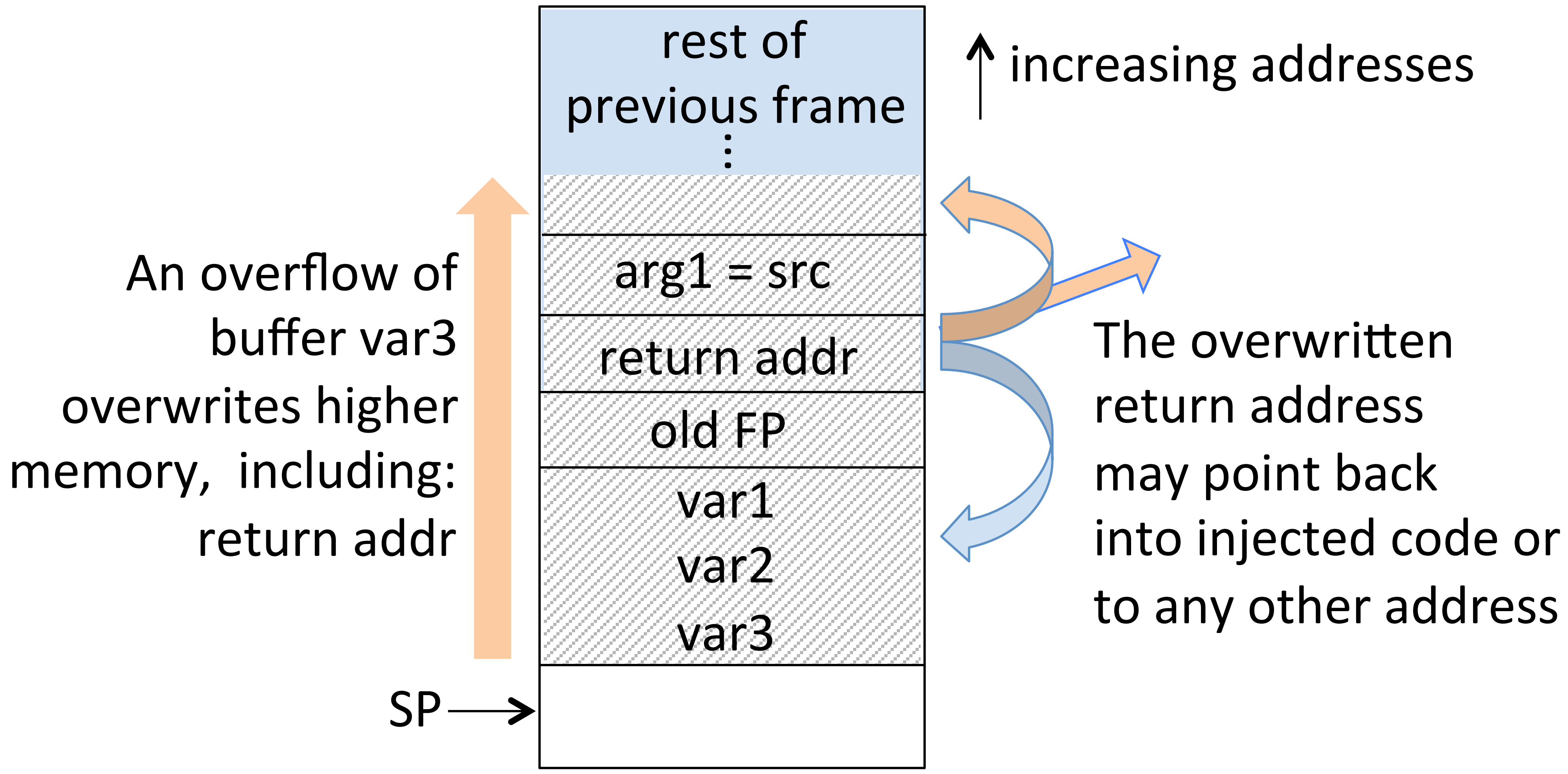
*Figure from Oorschot’s book

Eternal War in Memory
Round 1

Attack: Stack-based Buffer Overwrite

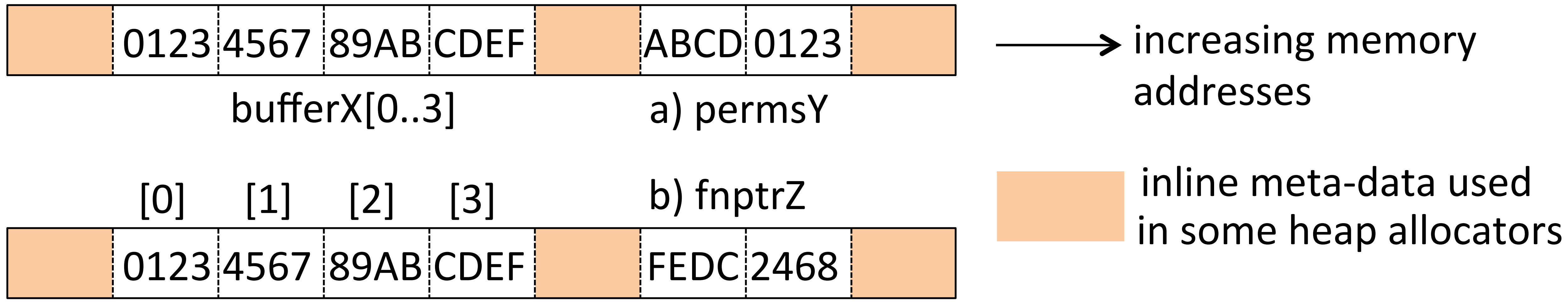
- Buffer overflows on stack can overwrite higher memory
    - Esp., return address
- Why is return address important?
    - Control-flow hijacking!

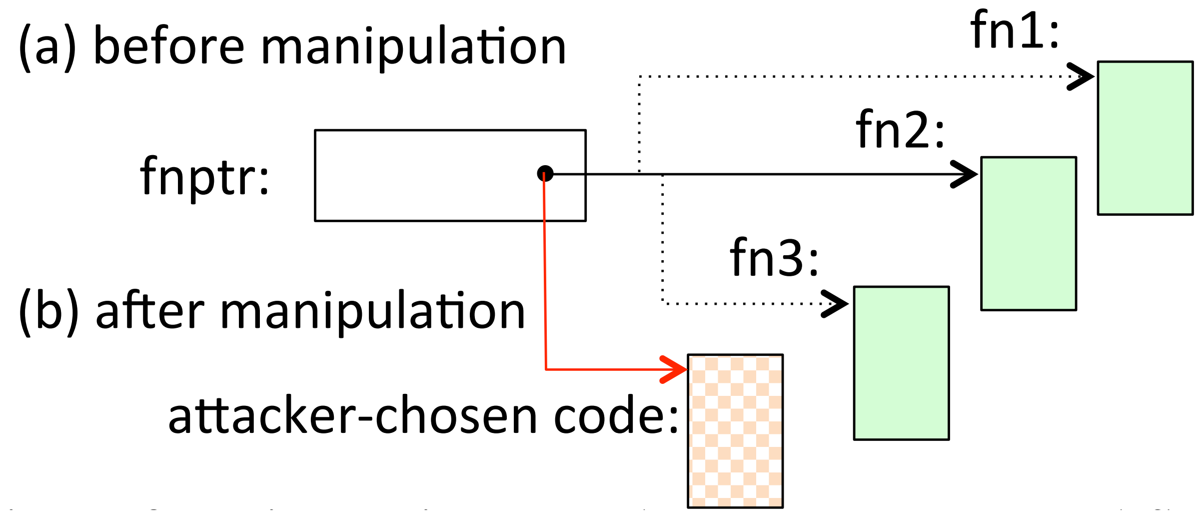
```
void myfunction(char *input) {
    int var1, var2;
    char var3[4];
    // what if the length of input > 3?
    strcpy(var3, input);
}
```



Attack: Heap-based Buffer Overwrite

- Find an exploitable buffer and a strategically useful variable for an attack
- Corrupt important data such as access control or function pointers





Effect of Buffer Overwrites

- Program control-flow can be directly altered by corrupting data
  - stack-based pointers (e.g., return addresses, frame pointers)
  - function pointers, jump table, etc
  - addresses used in setjmp/longjmp
  - (indirectly) by curating data used in a branching test
- DO use bound checking APIs!
  - E.g., strncpy, gets_s, scanf_s

Generic Exploit Steps

1. **Find** vulnerable locations in the target program

2. **Inject or locate code** that the attacker desires to be executed within the target program's address space

3. **Corrupt** control flow data (e.g., by a buffer overflow)

4. **Transfer** program control flow to the target code of step 2

Defense: Canary

- Idea: insert a random value (chosen at program start) before the return address
- Example:

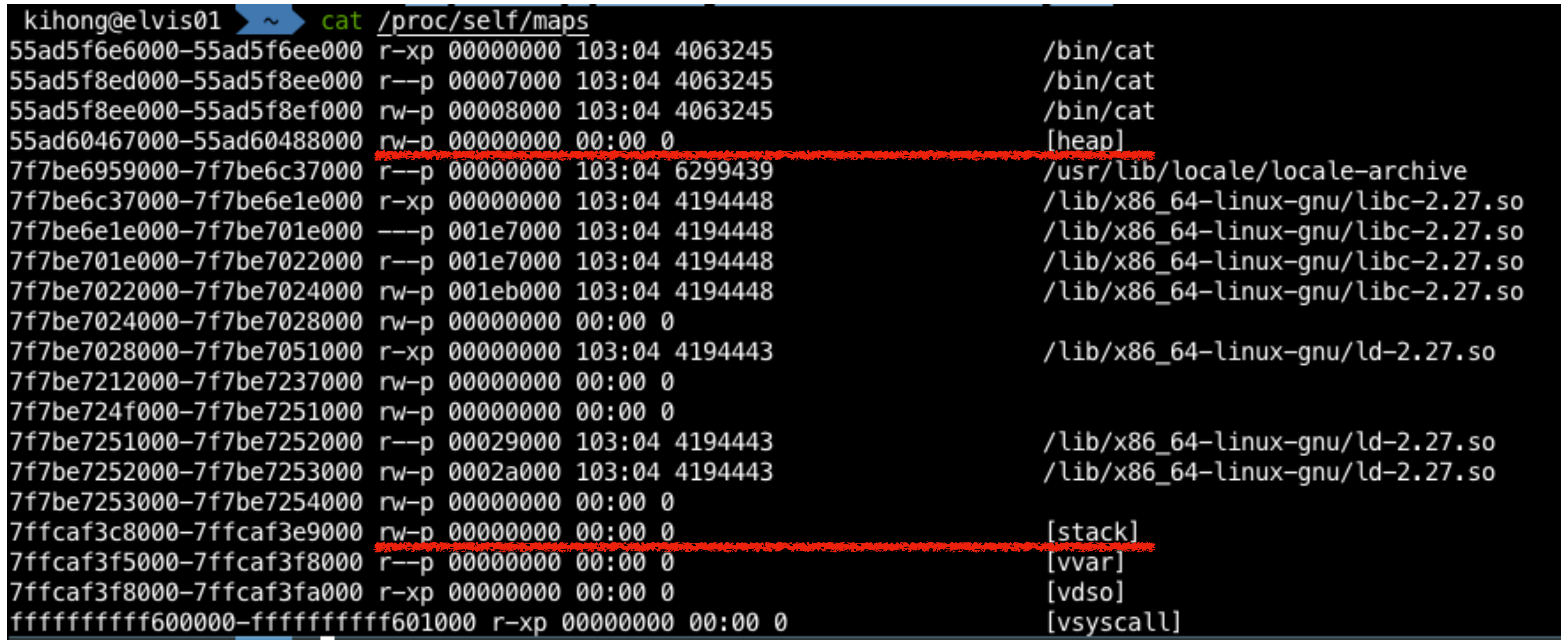
```c
void myfunction(char *input) {
    int var1, var2;
    char var3[4];
    // what if the length of input > 3?
    strcpy(var3, input);
}
```

```
// gcc -fstack-protector
myfunction:
    ...                        ; prologue
    mov ebx, gs:0x14           ; copy canary
    mov -0x4(ebb), ebx         ; insert canary
    xor ebx, ebx               ; clear register
    ...                        ; function body
    mov ebx, -0x4(ebx)         ; get local canary
    xor ebx, gs:0x14           ; check canary
    je <epilogue>              ; if so, return
    call <__stack_chk_fail> ; o.w., fail
    ...                        ; epilogue
```

…
return addr
old FP
stack canary
local vars

Defense: Non-executable Memory

- Data Execution Prevention (DEP)
- Certain address ranges are marked invalid for execution by OS or hardware
  - E.g., stack, heap, BSS, etc
- Caveat: not always applicable (backwards compatibility, use of JIT)



```
kihong@elvis01 ~ cat /proc/self/maps
55ad5f6e6000-55ad5f6ee000 r-xp 00000000 103:04 4063245                    /bin/cat
55ad5f8ed000-55ad5f8ee000 r--p 00007000 103:04 4063245                    /bin/cat
55ad5f8ee000-55ad5f8ef000 rw-p 00008000 103:04 4063245                    /bin/cat
55ad60467000-55ad60488000 rw-p 00000000 00:00 0                           [heap]
7f7be6959000-7f7be6c37000 r--p 00000000 103:04 6299439                    /usr/lib/locale/locale-archive
7f7be6c37000-7f7be6e1e000 r-xp 00000000 103:04 4194448                    /lib/x86_64-linux-gnu/libc-2.27.so
7f7be6e1e000-7f7be701e000 ---p 001e7000 103:04 4194448                    /lib/x86_64-linux-gnu/libc-2.27.so
7f7be701e000-7f7be7022000 r--p 001e7000 103:04 4194448                    /lib/x86_64-linux-gnu/libc-2.27.so
7f7be7022000-7f7be7024000 rw-p 001eb000 103:04 4194448                    /lib/x86_64-linux-gnu/libc-2.27.so
7f7be7024000-7f7be7028000 rw-p 00000000 00:00 0
7f7be7028000-7f7be7051000 r-xp 00000000 103:04 4194443                    /lib/x86_64-linux-gnu/ld-2.27.so
7f7be7212000-7f7be7237000 rw-p 00000000 00:00 0
7f7be724f000-7f7be7251000 rw-p 00000000 00:00 0
7f7be7251000-7f7be7252000 r--p 00029000 103:04 4194443                    /lib/x86_64-linux-gnu/ld-2.27.so
7f7be7252000-7f7be7253000 rw-p 0002a000 103:04 4194443                    /lib/x86_64-linux-gnu/ld-2.27.so
7f7be7253000-7f7be7254000 rw-p 00000000 00:00 0
7ffcaf3c8000-7ffcaf3e9000 rw-p 00000000 00:00 0                           [stack]
7ffcaf3f5000-7ffcaf3f8000 r--p 00000000 00:00 0                           [vvar]
7ffcaf3f8000-7ffcaf3fa000 r-xp 00000000 00:00 0                           [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                   [vsyscall]
```

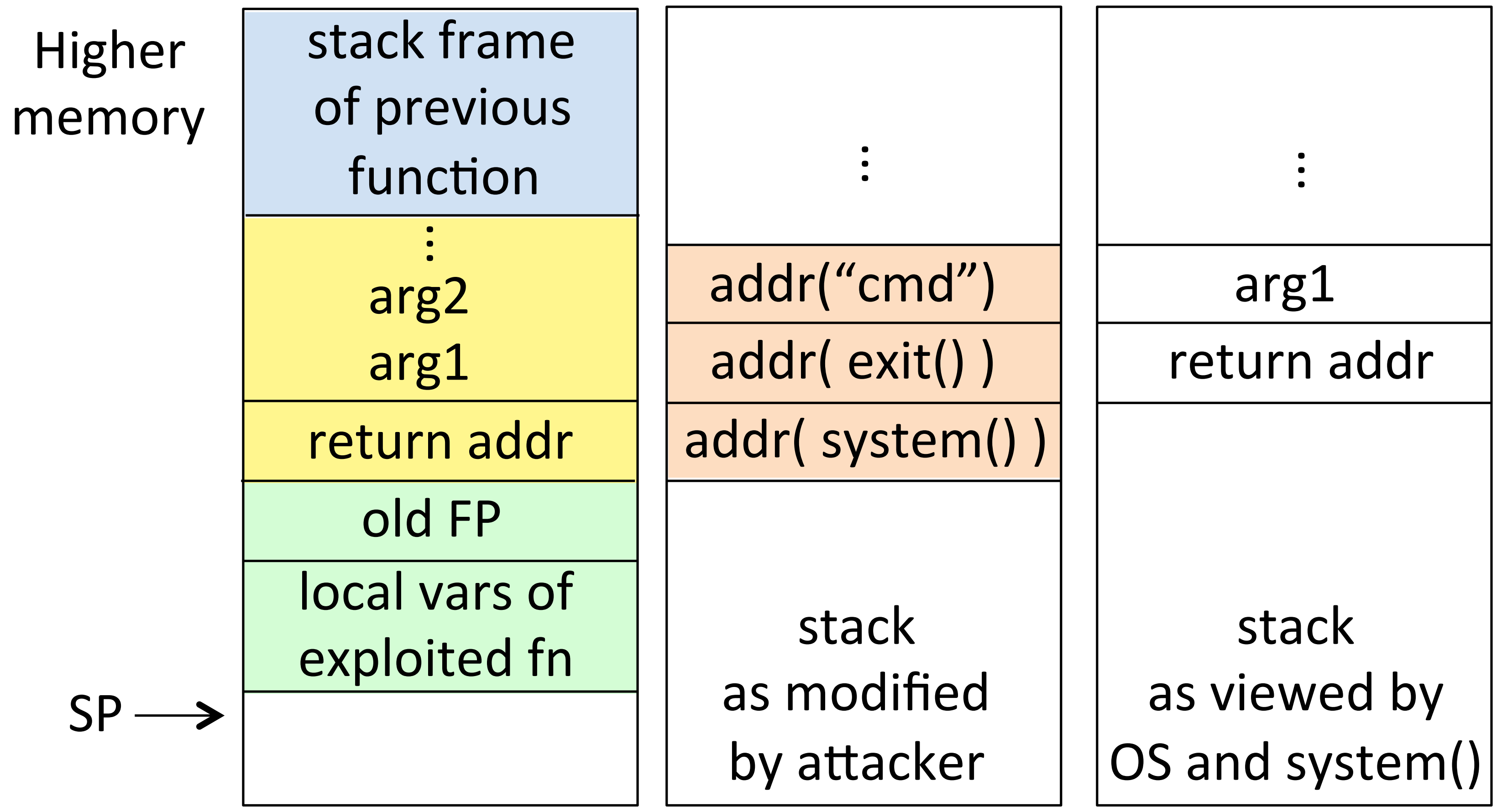
Summary of Round 1

- Buffer overwrite: attack by code injection + control-flow hijacking
    - Target: return address, function pointer, etc
- Code injection can be prevented by various defense techniques
    - E.g., Canary, Non-executable
- Are they enough to prevent all buffer overflow attacks?
- Why not use existing code?

Eternal War in Memory
Round 2

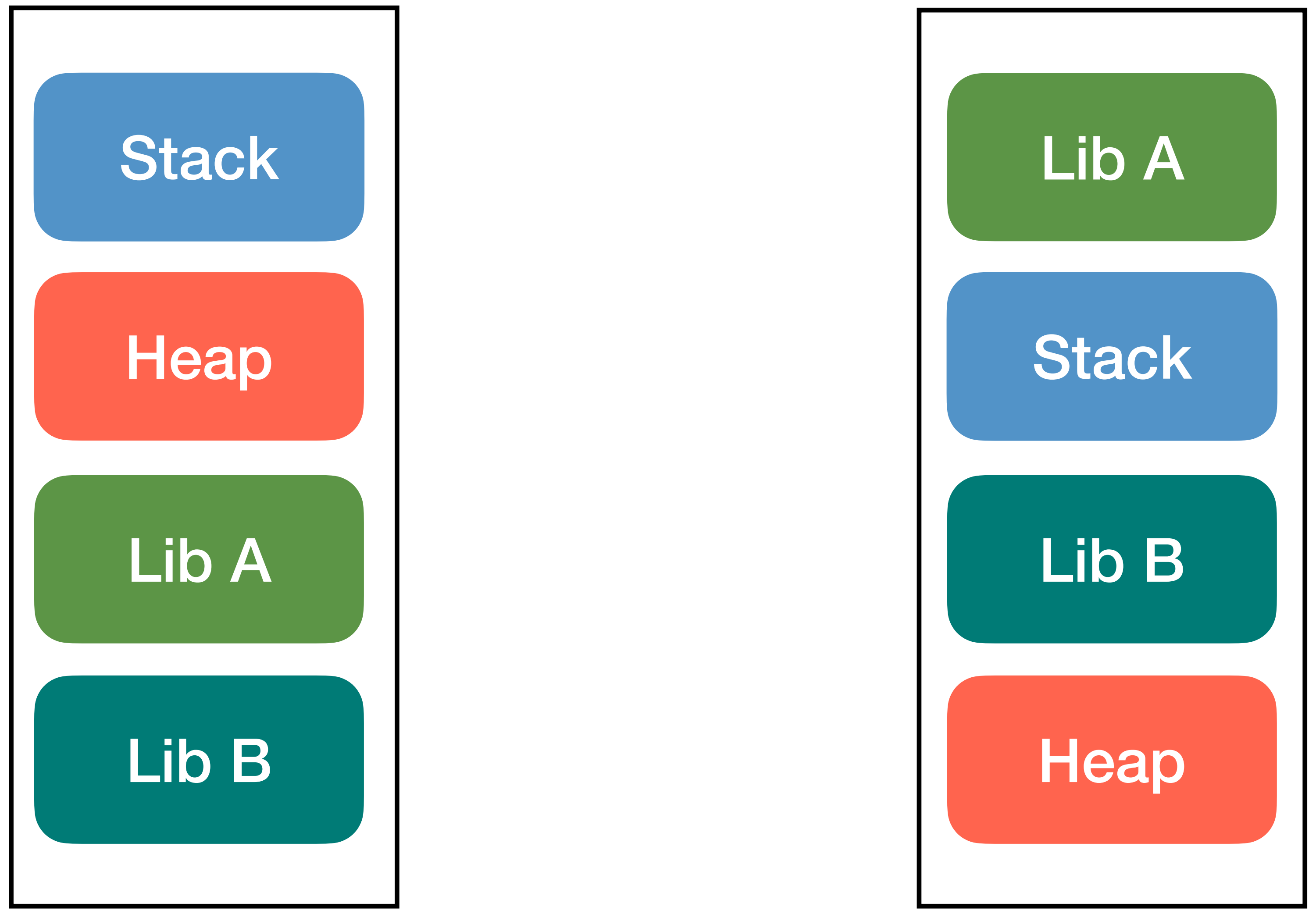
Attack: Return-to-libc

- Instead of injecting code into stack or heap, pass to existing system code
  - E.g., system calls or standard library functions in libc
- For example, pass code to `system()`



Defense: ASLR

- Address Space Layout Randomization
- Randomize the base address of the stack, heap, code, etc
  - E.g., randomly assign the base address of the segment of `libc`
- Introduced by Linux's PaX project, now in many mainstream OSs



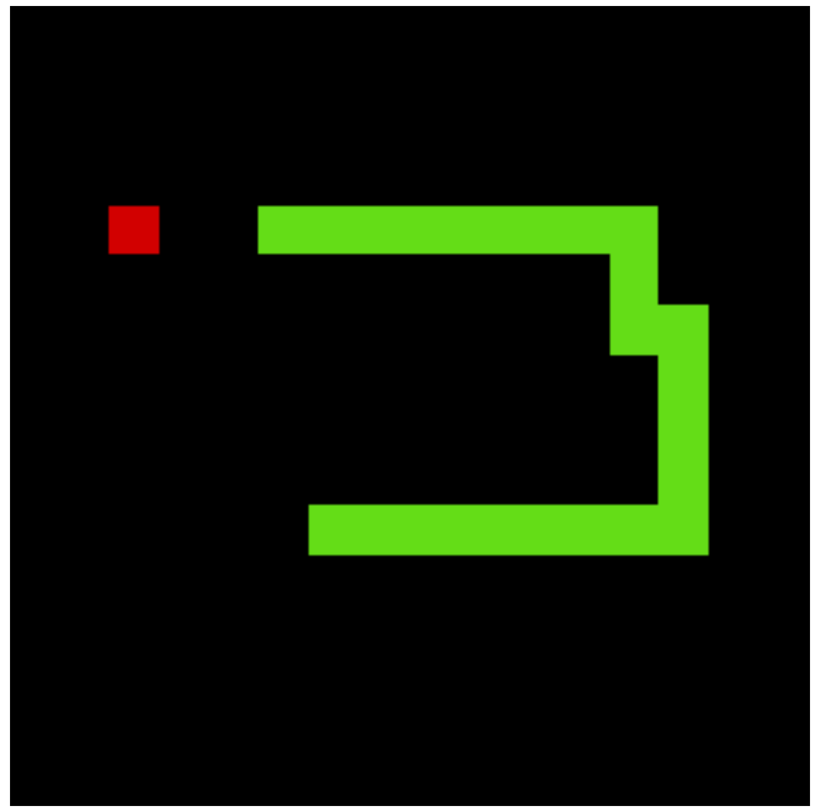
Summary of Round 2

- Attack: return-to-libc
    - Control-flow hijacking + code reuse
    - E.g., system()
- Defense: ASLR
    - Hide the address of desired libc code
- Attack: brute force search
    - Try all possibilities (easy for 32-bit machines!)
- Defense: avoid using libc

Eternal War in Memory
Round 3

Attack: Return-oriented Programming

- Instead of using a single libc function, use a sequence of existing code called **gadgets**
- Gadget: binary code snippets that end with a jump instruction (e.g., `ret`, `jmp`, etc)
  - ROP Gadgets: instruction sequence that ends with `ret`
  - Jump to another gadget via `ret`
- Exploitation steps: 1) find the gadgets you need and 2) stitching them together



Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

	0x8830
	0x4000
	0x8810
	0x8820
	0x0002
	0x8810
	0x0001
esp →	0x8800

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
0x8810
0x8820
0x0002
0x8810
esp → 0x0001

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
0x8810
0x8820
0x0002
esp → 0x8810

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
0x8810
0x8820
esp → 0x0002

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
0x8810
esp → 0x8820

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
esp → 0x8810

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000
esp → 0x8810

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

0x8830
0x4000

esp → 0x4000

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention 

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

```
esp → 0x8830
```

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Example

- Suppose an attacker smashed the stack and made the program to execute `ret`

Stack (smashed)

Code (existing)

```
0x8830:
  mov [ebx], eax
  ret
  …
0x8820:
  add eax, ebx
  ret
  …
0x8810:
  pop ebx
  ret
  …
0x8800:
  pop eax
  ret
```

Intention 

```
eax = 1

ebx = 2

eax += ebx

ebx = 0x4000

*ebx = eax
```

Summary of Round 3

- Attack: return-oriented programming (ROP)
    - Control-flow hijacking + finer-grained code reuse via gadget
- Defense:
    - Control-flow integrity: estimate then allow valid control-flows only
    - Program debloating: remove unnecessary code
- More attack & defense
- Eternal War in Memory

Case Study: Integer-based Vulnerabilities

- Exploitable code sequences due to integer bugs
  - E.g., unsafe type casting, integer overflow, etc

```
BOOL handle_login(userid, password) {
  attempts = attempt + 1;
  // what if “attempts” overflows?
  if (attempts <= MAX_ALLOWED) {
    if (pswd_is_ok(userid, password) {
      attempts = 0;
      return TRUE;
    }
  }
  return FALSE;
}
```

```
void init_table() {
  unsigned int width = input();
  unsigned int height = input();
  // what if “width * height” overflows?
  table = malloc(width * height);
  for (i = 0; i < width; i++) {
    for (j = 0; j < height; j++) {
      … table[i][j] …
}
```

Consequences of Integer Vulnerabilities

- Unexpected subscript: enable access to unintended addresses
- Under-allocation of memory: smaller than anticipated integer values
- Out-of-memory: neg size-arg to malloc → large pos integer (underflow)
- Excessive number of iterations: overflow → large neg integer compared to an upper bound of a loop
- Etc

Case Study: Use-After-Free

- malloc: allocate a memory block on the heap
    - Find an appropriate block from the list of free blocks
- free: release the allocated memory block
    - Return the block to the free list
    - Typically do not erase the contents for efficiency
- What happens if a block is used after free?

```
ptr = malloc(…); //0xabcd1234
login(ptr);
free(ptr);
// use of ptr
```

0xabcd1234

user: admin password: $3cret!

Towards Safe & Reliable SW Development

- Basic SW engineering principles
  - E.g., formatting, testing, and clean code
- Secure coding
- Safe programming languages
  - E.g., OCaml, Scala, Rust, etc
- Advanced software analysis tools

THE WHITE HOUSE

Administration The Record The Issues Briefing Room Español MENU

FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe

ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Read the full report here

WASHINGTON – Today, the White House Office of the National Cyber Director (ONCD) released a report calling on the technical community to proactively reduce the attack surface in cyberspace. ONCD makes the case that technology manufacturers can prevent entire classes of vulnerabilities from entering the digital ecosystem by adopting memory safe programming languages. ONCD is also encouraging the research community to address the problem of software measurability to enable the development of better diagnostics that measure cybersecurity quality.

The report is titled *"Back to the Building Blocks: A Path Toward Secure and Measurable Software."*

*White House, 2024

Summary

- Software security can affect physical & data security
  - SW can manipulate machines and read / write data
- SW bugs can lead to security problems
- Growing interest as SW is eating the world!
  - Traditional SW: financial, military, privacy, etc
  - Emerging concerns: security of AI such as fairness or morality