CSE509 : Computer System Security

Memory Error Exploits and Defenses

Process Memory Layout

high mem

Argv/Env: CLI args and environment
Stack: generally grows downwards
Heap: generally grows upwards
BSS: unitialized global data
Data: initialized global data
Text: read-only program code

low mem

Memory Layout Example

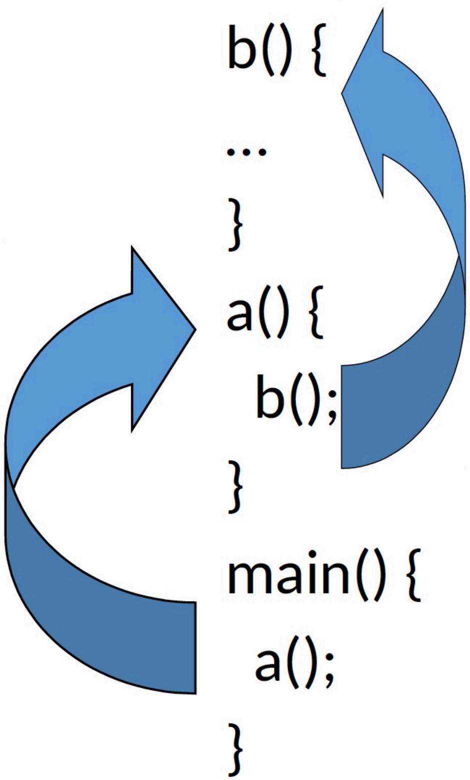
```
/* data segment: initialized global data */
int a[] = { 1, 2, 3, 4, 5 };
/* bss segment: uninitialized global data */
int b;
/* text segment: contains program code */
int main(int argc, char **argv) /* ptr to argv */
{
        /* stack: local variables */
        int *c;
        /* heap: dynamic allocation by new or malloc */
        c = (int *)malloc(5 * sizeof(int));
}
```

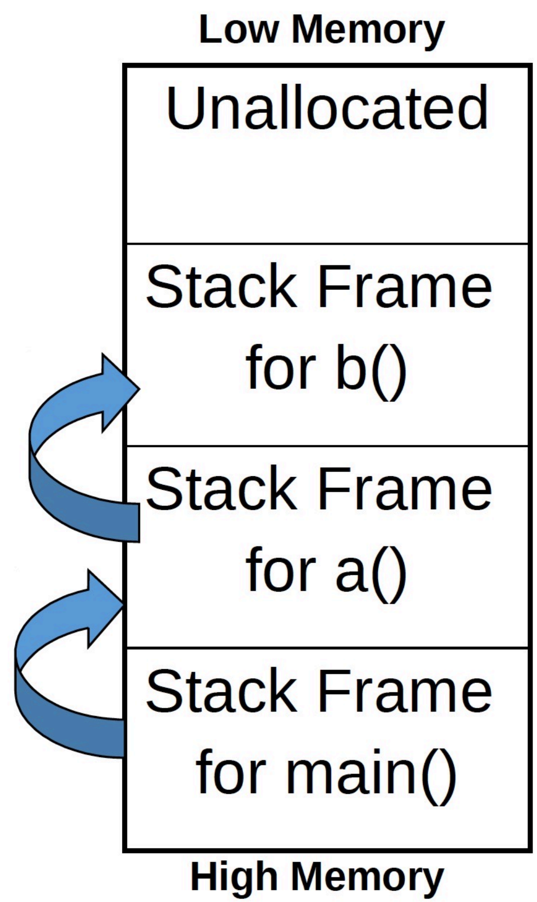
What is the Call Stack?

- LIFO data structure: push/pop
  - Stack grows downwards in memory.
  - SP (esp) points to top of stack (lowest address)

  - What's on the call stack?
    - Function parameters
    - Local variables
    - Return values
    - Return address

Call Stack Layout





Accessing the Stack

- Pushing an item onto the stack.
  1. Decrement SP by 4.
  2. Copy 4 bytes of data to stack.

  Example: push 0x12
- Popping data from the stack.
  3. Copy 4 bytes of data from stack.
  4. Increment SP by 4.

  Example: pop eax

  Retrieve data without pop: mov eax, esp

What is a Stack Frame?

- Block of stack data for one procedure call.
- Frame pointer (FP) points to frame:
  - Use offsets to find local variables.
  - SP continually moves with push/pops.
  - FP only moves on function call/return.
  - Intel CPUs use ebp register for FP

C Calling Convention

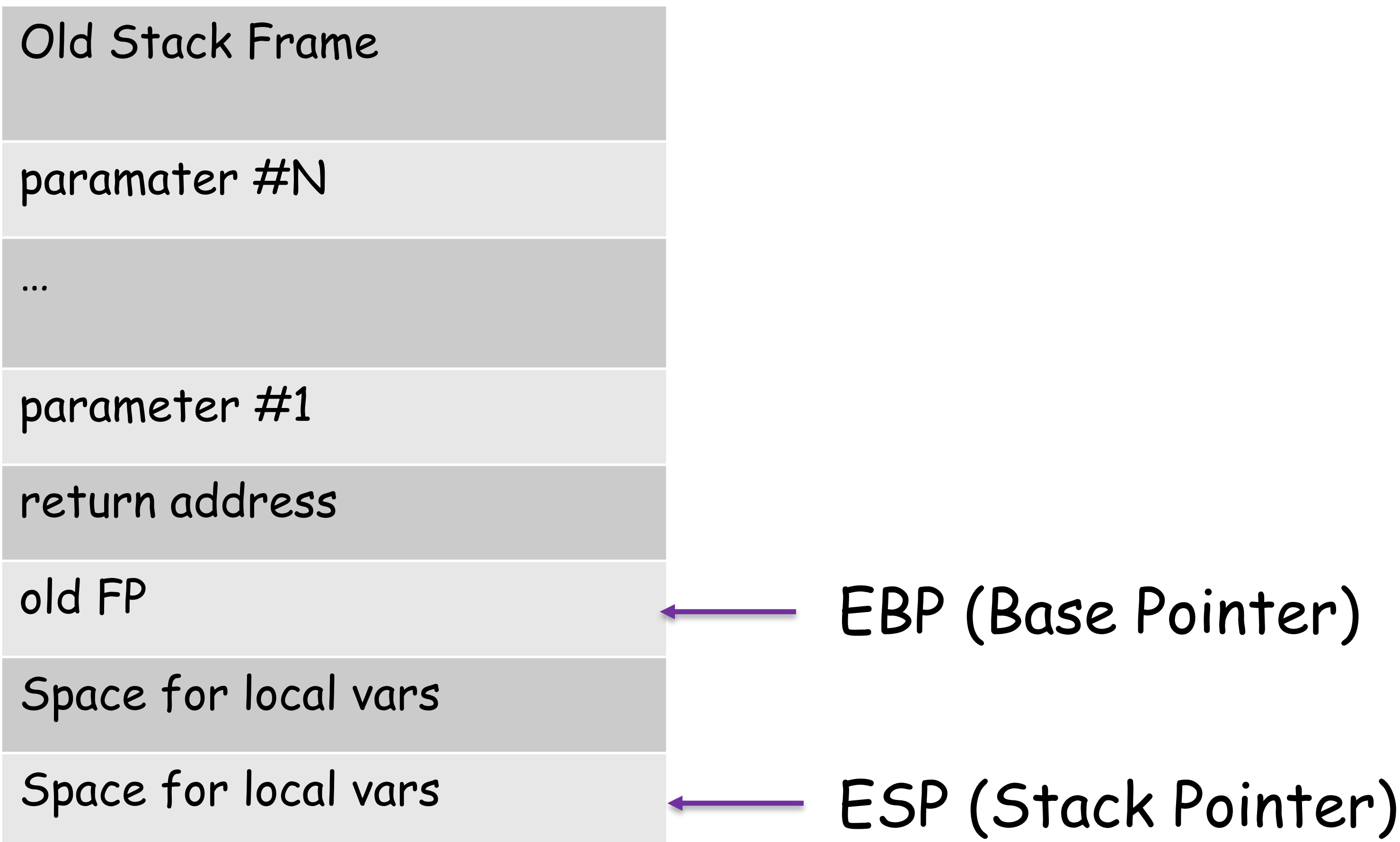
1. Push all params onto stack in reverse order.
   - Parameter #N
   - ...
   - Parameter #2
   - Parameter #1
2. Issues a call instruction.
   1. Pushes address of next instruction (the return address) onto stack.
   2. Modifies IP (eip) to point to start of function.

Stack before Function Executes

C Calling Convention

1. Function pushes FP (ebp) onto stack.
   Save FP for previous function.
   push ebp
2. Copies SP to FP.
   Allows function to access params as fixed indexes from base pointer.
   mov ebp,esp
3. Reserves stack space for local vars.
   subl esp, 0x12

Stack at Function Start



C Calling Convention

1. After execution, stores return value in eax.

   movl eax, 0x1

   Resets stack to pre-call state.

   Destroys current stack frame; restores caller's frame.

   mov esp, ebp

   pop ebp

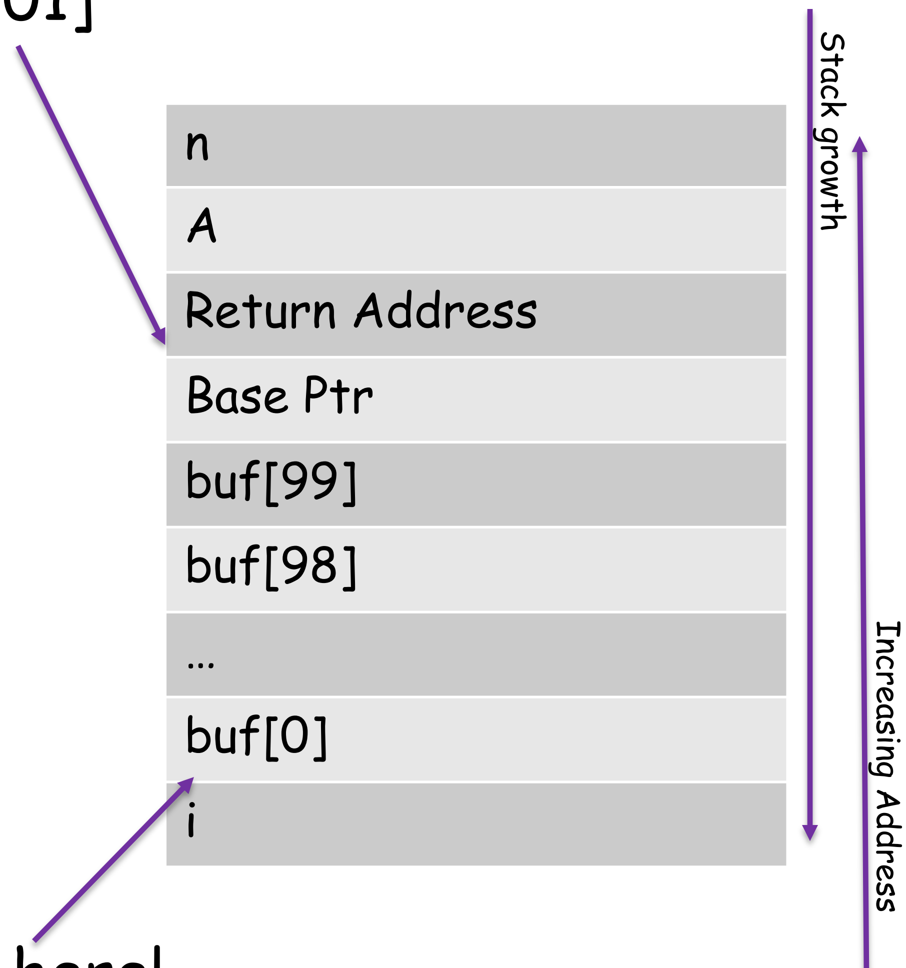
2. Returns control back to where called from.

   ret pops top word from stack and sets eip to that value.

Example: Stack Smashing Attack

```
void
f(const int *A, int n) {
    int buf[100];
    int i = 0;
    while (i < n) {
        buf[i] = A[i++];
    }
    ...
}
```

buf[101]



Injected code starts here!

Stack smashing defenses

- Canary stored before return value, checked before return
  - Issues
    - Protecting RA vs Saved BP
    - Random, XOR, null Canaries
    - How about data?
  - Weaknesses
    - Brute-force Canary, or rely on information leakage attacks
    - Overwrite RA without Overwriting canary (e.g., double pointer attacks)
    - Overwrite other code pointers (e.g., function pointer, virtual table pointer, GOT)
- Storing RA in two places
  - StackShield, Return Address Defender (RAD)
  - Issues: Compatibility with signals, exceptions, longjmp

Stack smashing defenses

- Propolice
  - Canary before saved BP + protect local variables by reordering them
    - Simple variables (integers, pointers) located at lower addresses, buffers at higher addresses
      - Buffer overflow cannot corrupt local variables, preventing double pointer attacks
        - But underruns can corrupt these simple (non-buffer) variables
  - Mainstream compilers (gcc, MS) include Propolice like protection
    - Not included for functions with no arrays

Non-executable data

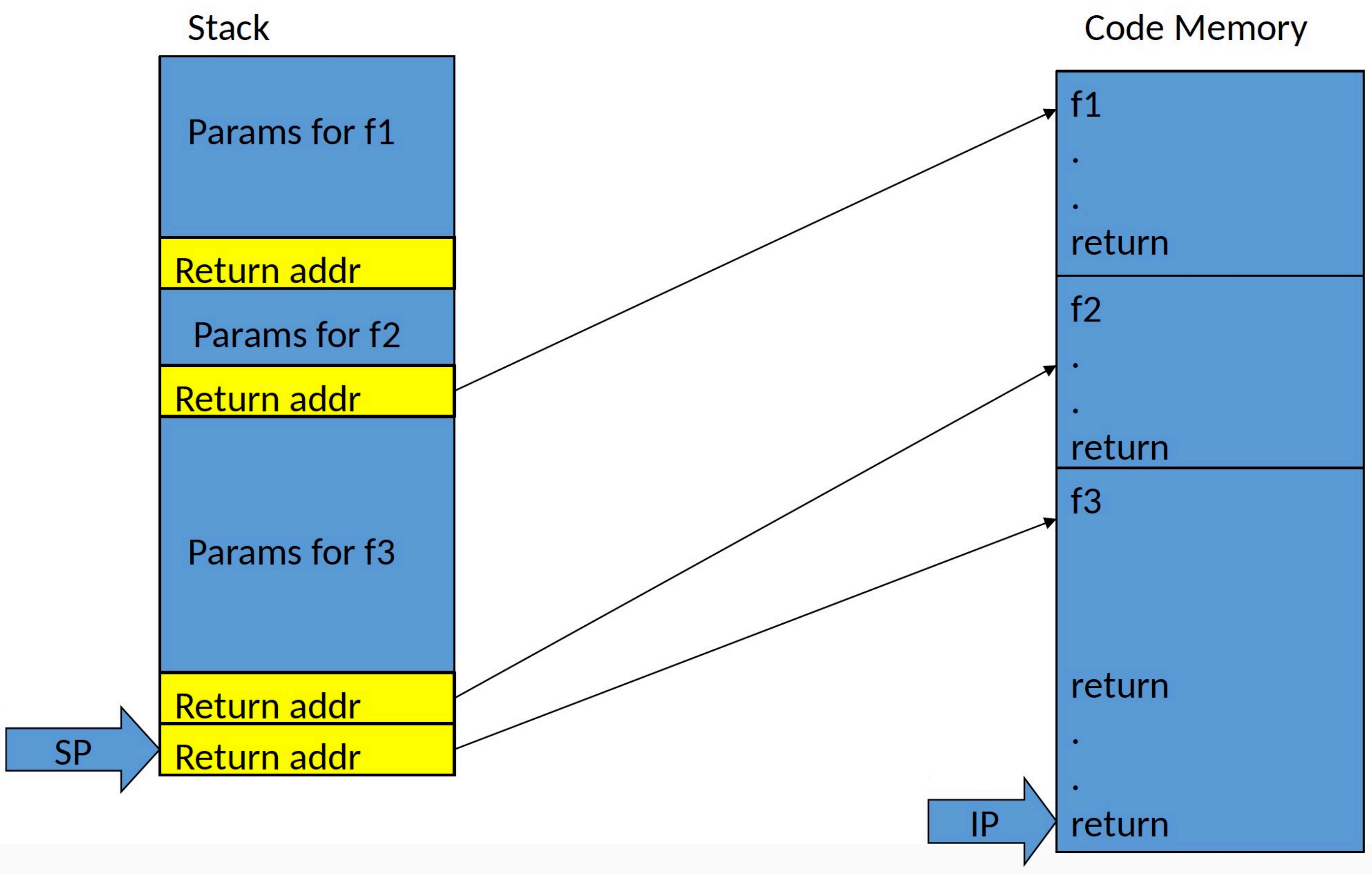
- Direct code injection attacks at some point execute data
  - Most programs never need to do this

    Hence, a simple countermeasure is to mark data memory (stack, heap, ...) as non-executable
  - Write-XOR-Execute, DEP
- This counters direct code injection\
  - In principle, this countermeasure may also break certain legacy applications

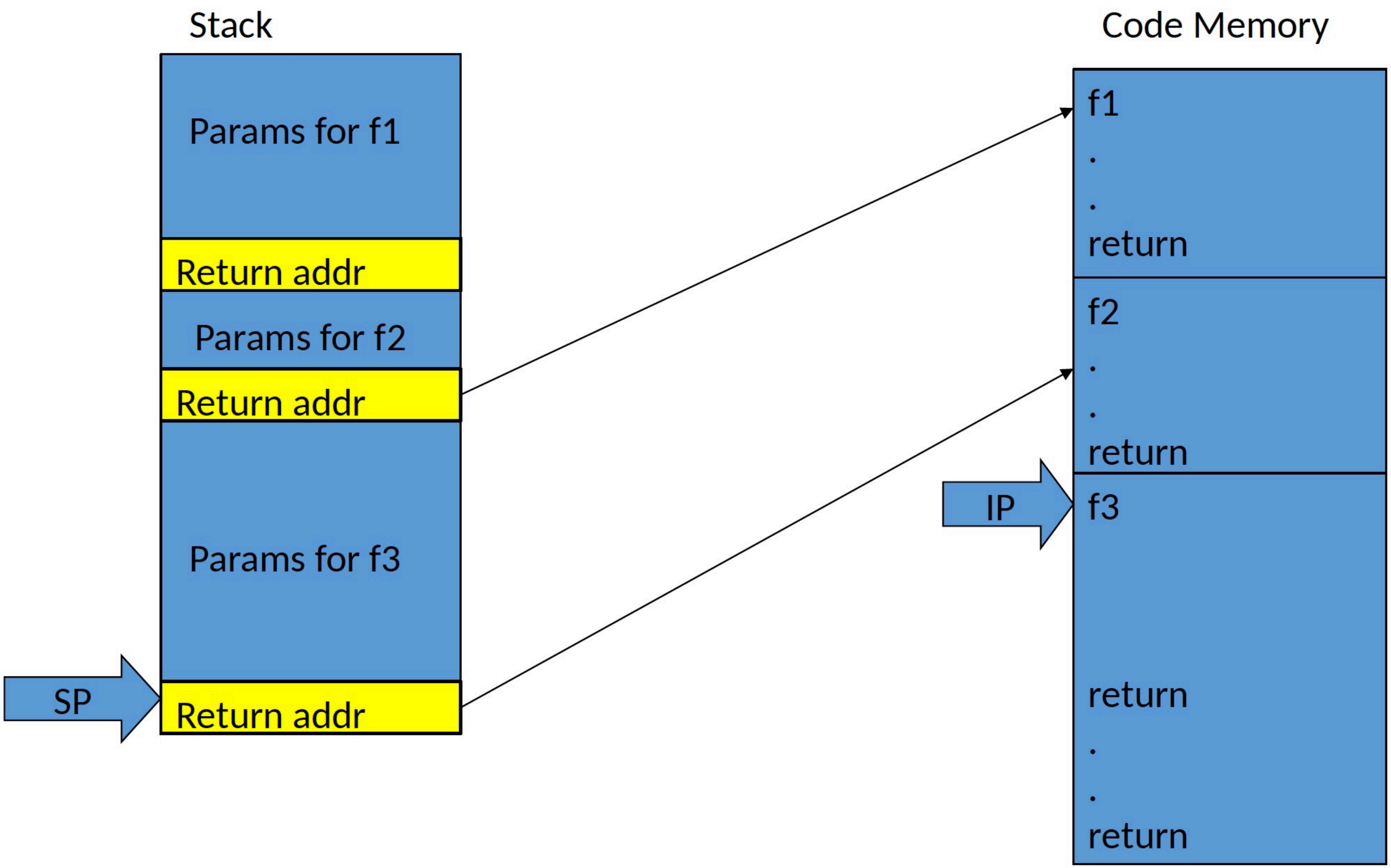
Reaction: No code injection necessary

- Instead of injecting malicious code, why not assemble malicious code out of existing code already present in the program
  - Indirect code injection attacks will drive the execution of the program by manipulating the stack
- E.g. Just execute system("/bin/bash") instead of creating your own interrupts
  - You just need to find where the system function is and call it with the right parameter

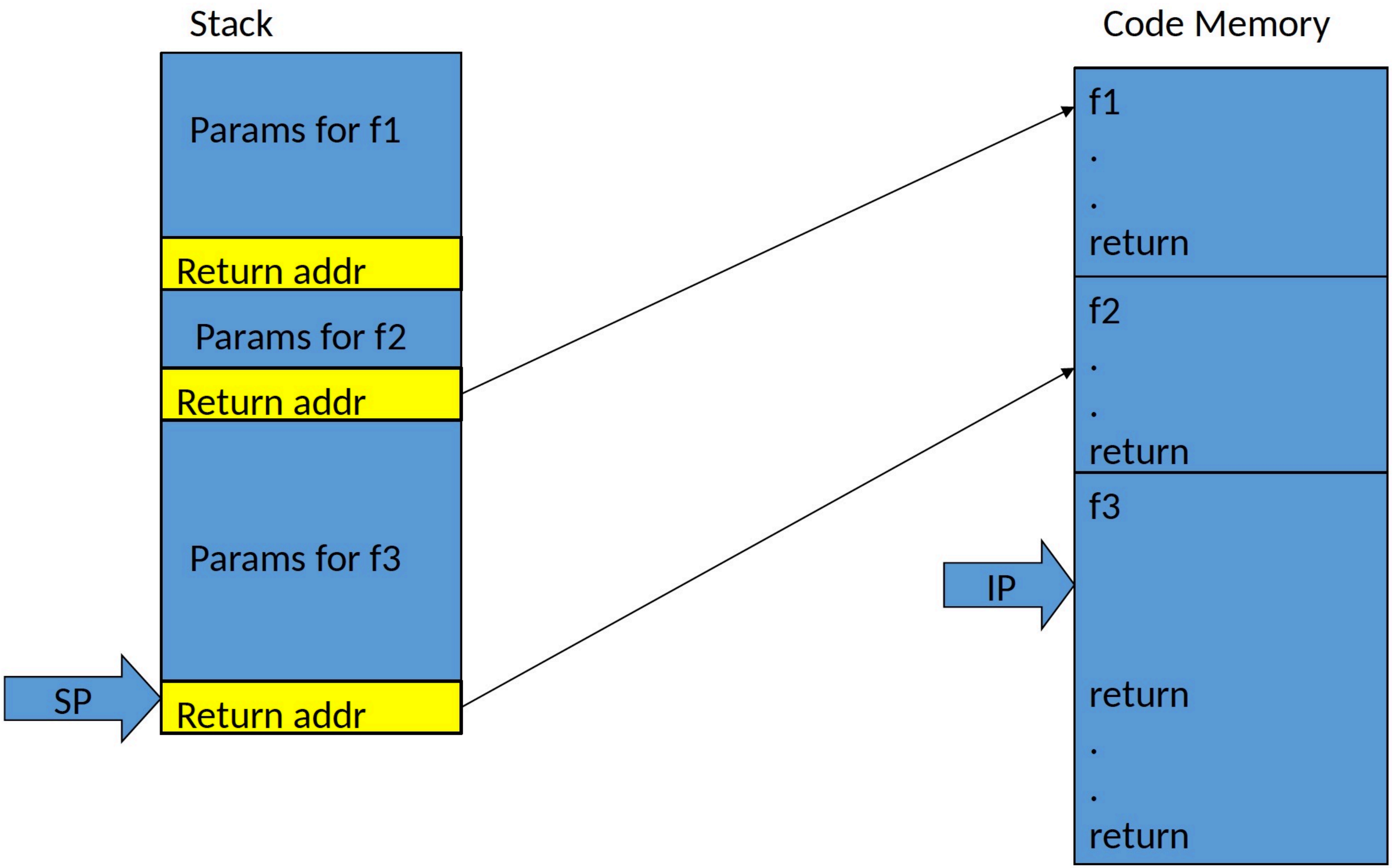
Return-into-libc: overview



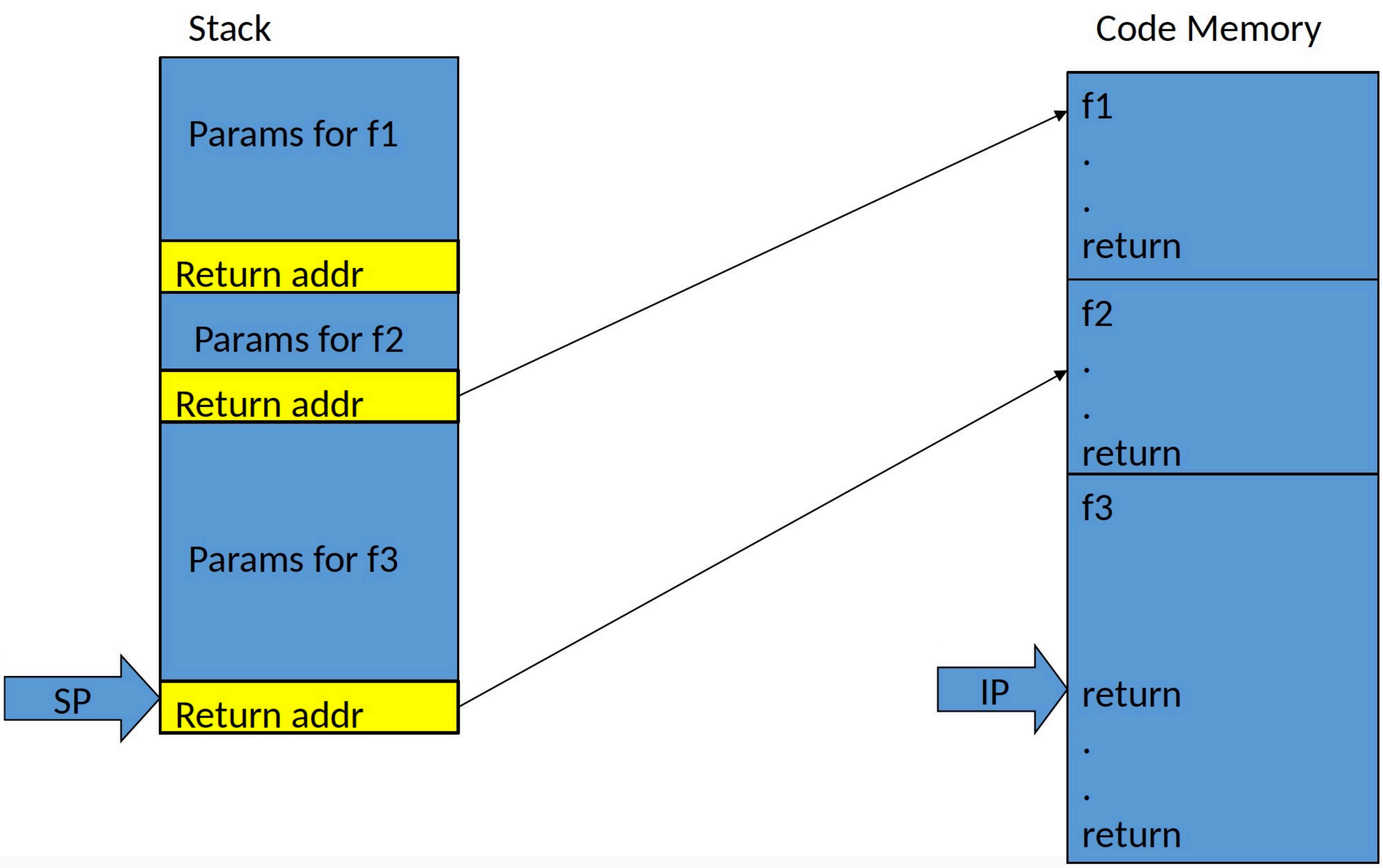
Return-into-libc: overview



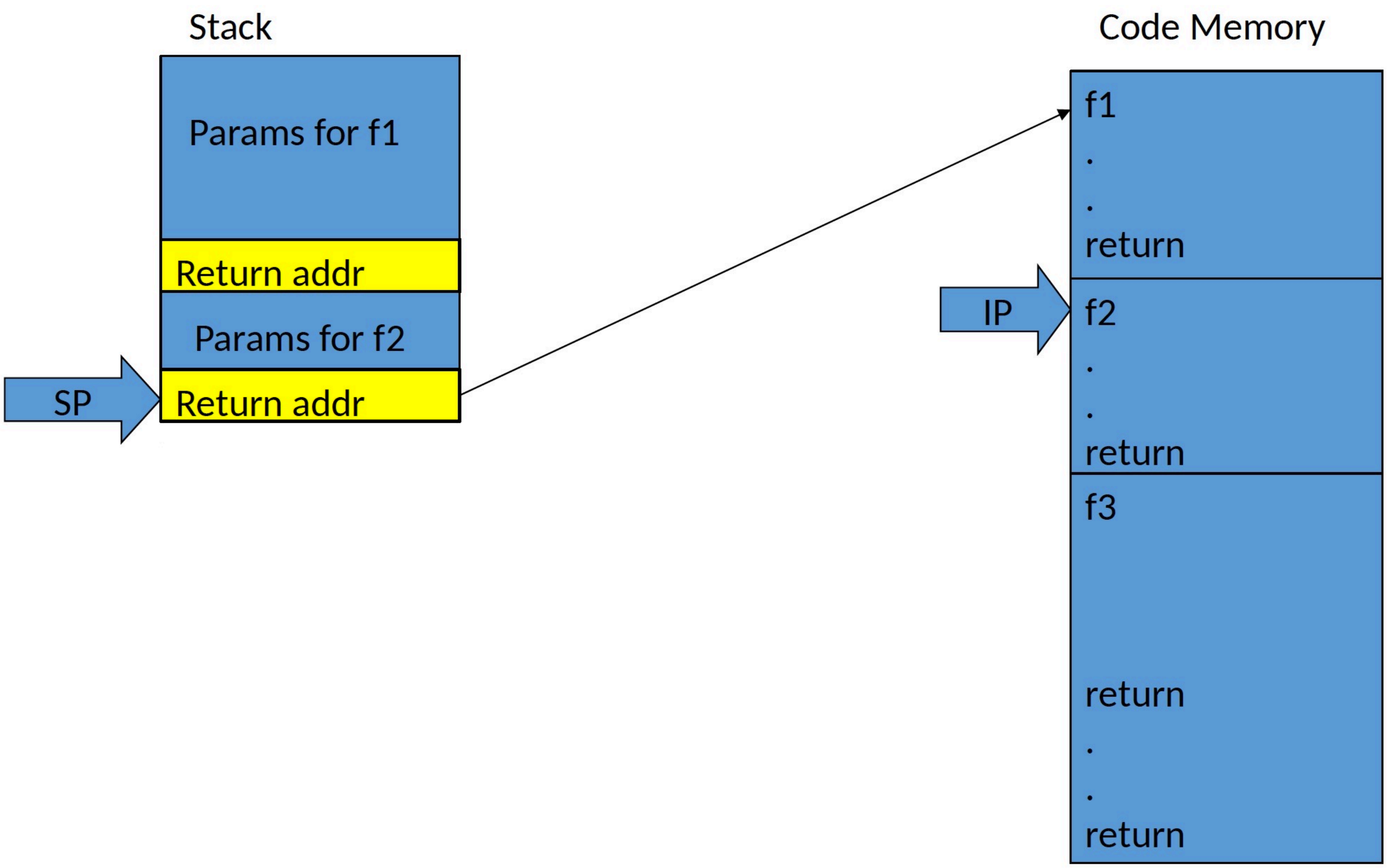
Return-into-libc: overview



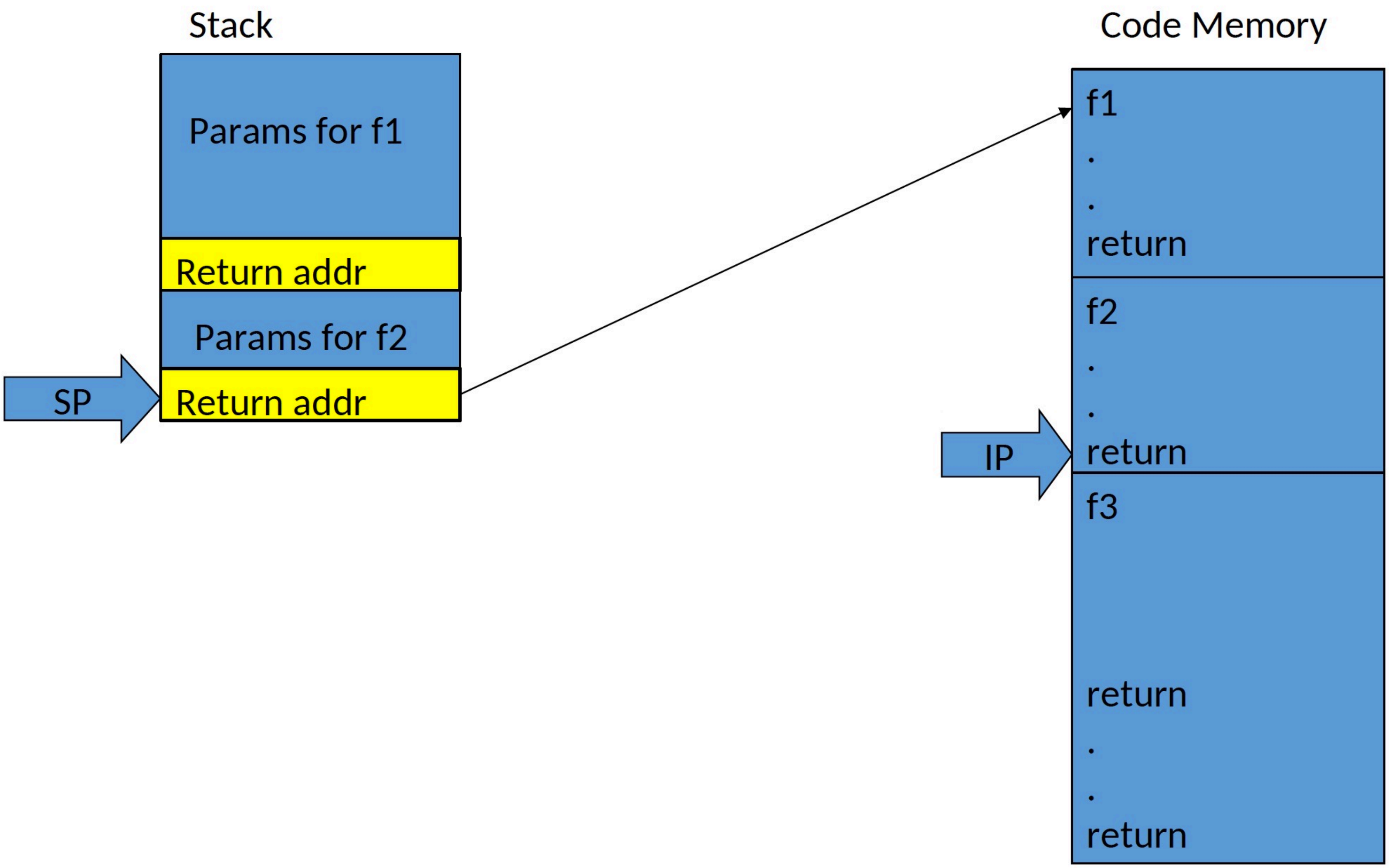
Return-into-libc: overview



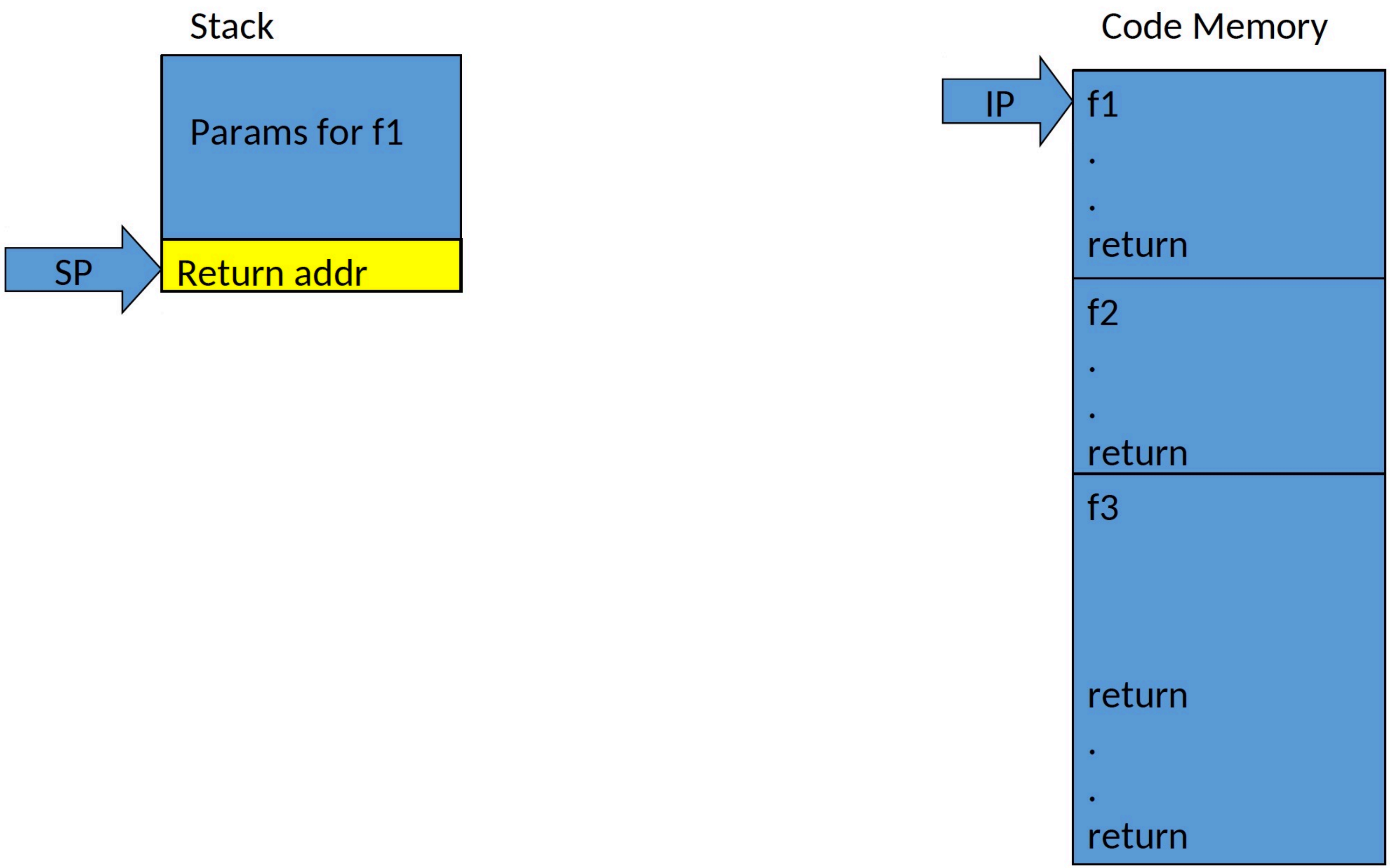
Return-into-libc: overview



Return-into-libc: overview



Return-into-libc: overview



Return-to-libc

- What do we need to make this work?
  - Inject the fake stack
    - Easy: this is just data we can put in a buffer

  - Make the stack pointer point to the fake stack right before a return instruction is executed

  - Then we make the stack execute existing functions to do a direct code injection
    - But we could do other useful stuff without direct code injection

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

- 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

High

...
...
...
0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

ESP → 0x80abcdee

Low

EAX = SMTH
EBX = SMTH
ECX = SMTH

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...0
x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

Return Oriented Programming

High

…
…
…
0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

Low

ESP

EIP

EAX = SMTH
EBX = SMTH
ECX = SMTH

…
0x80345677: pop $ecx;
0x80345678: ret;
…
0x08abcdee: pop $eax;
0x08abcdef : ret;
…0
x80abddaa: pop $ebx;
0x80abddab: ret;
…
0x80abdea0: int 0x80;
…

Return Oriented Programming

Stack (High → Low)
…
…
…
0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

High

Low

ESP →

EIP →

EAX = 8
EBX = SMTH
ECX = SMTH

…
0x80345677: pop $ecx;
0x80345678: ret;
…
0x08abcdee: pop $eax;
0x08abcdef : ret;
…0
x80abddaa: pop $ebx;
0x80abddab: ret;
…
0x80abdea0: int 0x80;
…

Return Oriented Programming

EAX = 8
EBX = SMTH
ECX = SMTH

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...0
x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

Return Oriented Programming

EAX = 8
EBX = &"/tmp..."
ECX = SMTH

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...0
x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

Return Oriented Programming

High

…
…
…
0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

ESP →

Low

EAX = 8
EBX = &"/tmp…"
ECX = SMTH

EIP →

```
…
0x80345677: pop $ecx;
0x80345678: ret;
…
0x08abcdee: pop $eax;
0x08abcdef : ret;
…0
x80abddaa: pop $ebx;
0x80abddab: ret;
…
0x80abdea0: int 0x80;
…
```

Return Oriented Programming

	...
	...
	...
ESP →	0x80abdea0
	0x309
	0x80345677
	&"/tmp/lala"
	0x80abddaa
	8
	0x80abcdee

High

Low

EAX = 8
EBX = &"/tmp..."
ECX = 0x309

EIP →

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...0
x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

Return Oriented Programming

ESP →

…	High
…	
…	
0x80abdea0	
0x309	
0x80345677	
&"/tmp/lala"	
0x80abddaa	
8	
0x80abcdee	
	Low

EAX = 8
EBX = &"/tmp..."
ECX = 0x309

```
...
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...0
x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

EIP → (0x80abdea0: int 0x80;)

Heap based buffer overflow

- If a program contains a buffer overflow vulnerability for a buffer allocated on the heap, there is no return address nearby
- So attacking a heap based vulnerability requires the attacker to overwrite other code pointers
- We look at two examples:
  - Overwriting a function pointer
  - Overwriting heap metadata

Overwriting a function pointer

- Example vulnerable program:

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Overwriting a function pointer

- And what happens on overflow:

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x662f2f3a	0x61626f6f	0x00000072	0x004013ce

(a) A structure holding "file://foobar" and a pointer to the `strcmp` function.

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x612f2f3a	0x61666473	0x61666473	0x00666473

(b) After a buffer overflow caused by the inputs "file://" and "asdfasdfasdf".

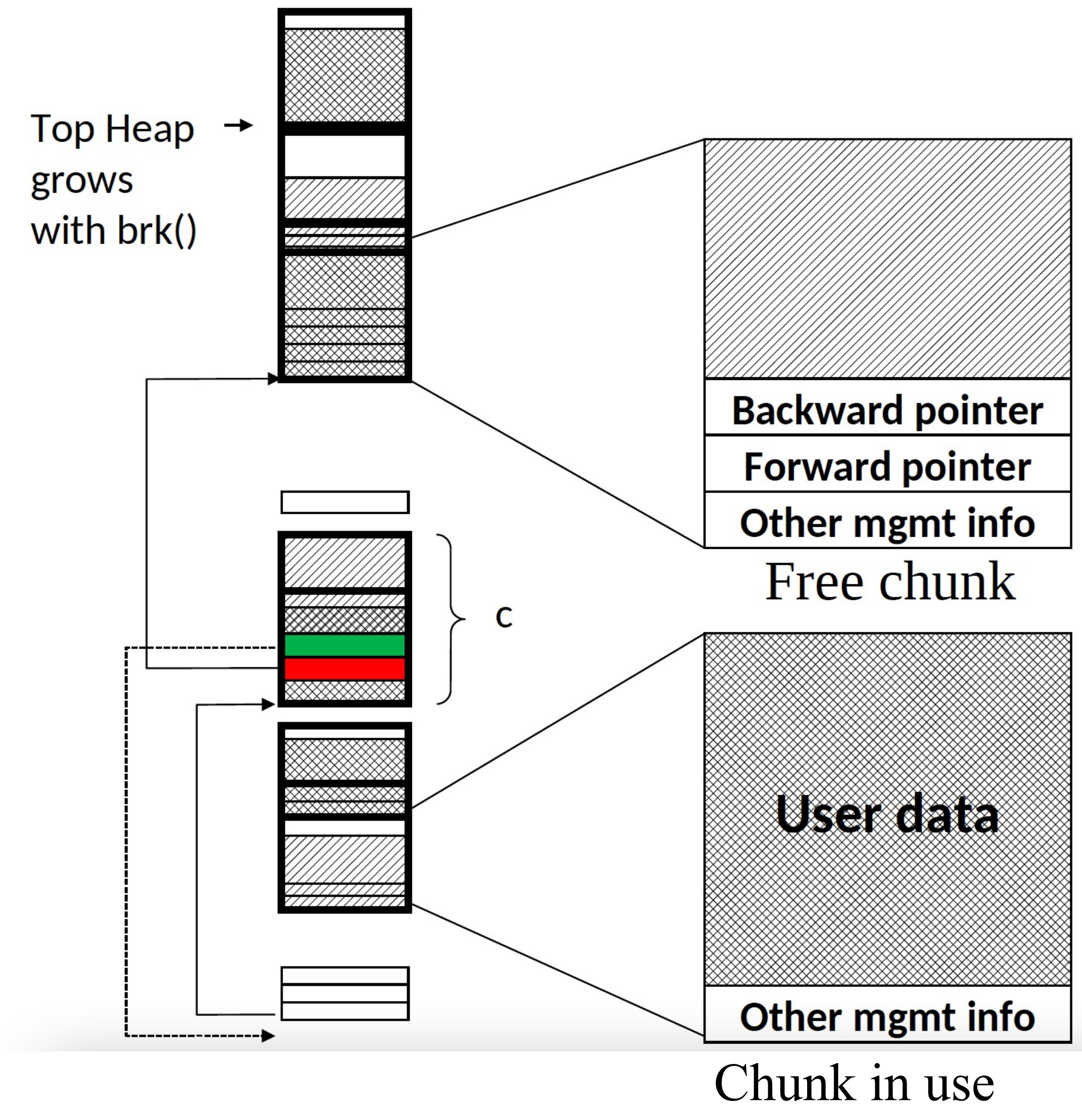
	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0xfeeb2ecd	0x11111111	0x11111111	0x11111111	0x00353068

(c) After a malicious buffer overflow caused by attacker-chosen inputs.

Overwriting heap metadata

- The heap is a memory area where dynamically allocated data is stored
  - Typically managed by a memory allocation library that offers functionality to allocate and free chunks of memory (in C: malloc() and free() calls)
- Most memory allocation libraries store management information in-band
  - As a consequence, buffer overruns on the heap can overwrite this management information
  - This enables an "indirect pointer overwrite"-like attack allowing attackers to overwrite arbitrary memory locations

Heap management in dlmalloc

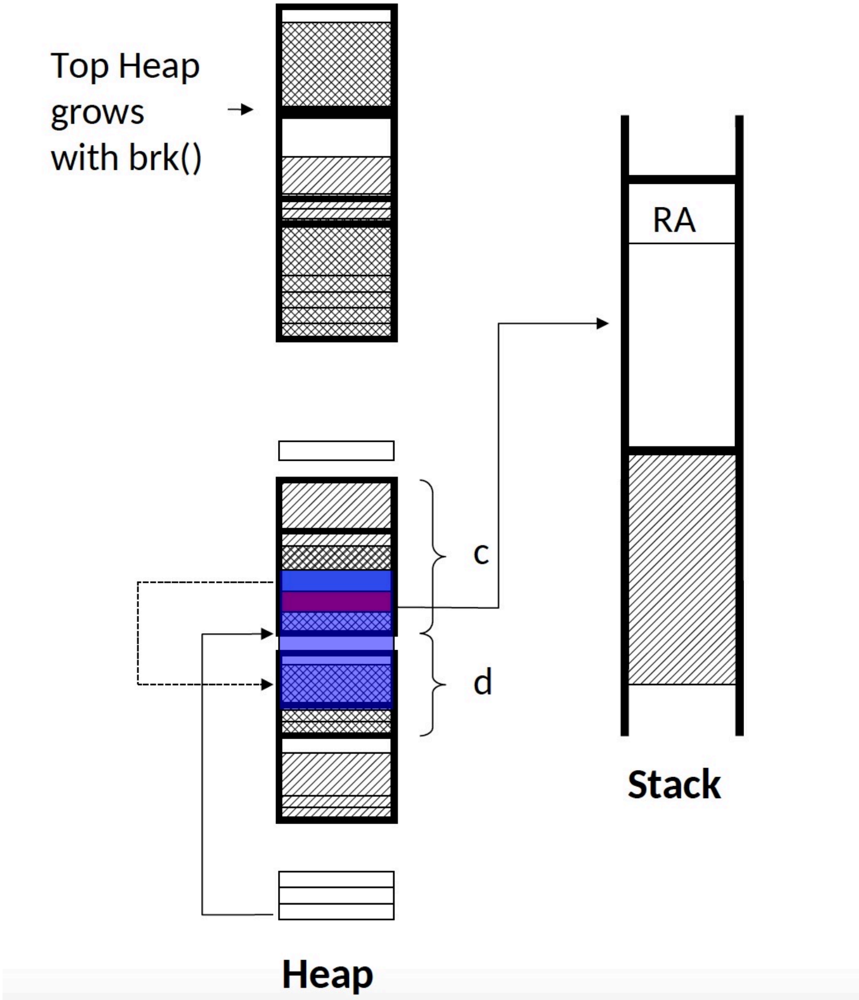


Dlmalloc maintains a doubly linked list of free Chunks

When chunk c gets unlinked, c's backward pointer is written to *(forward pointer+12)

Or: green value is written 12 bytes above where red value points

Exploiting a buffer overrun



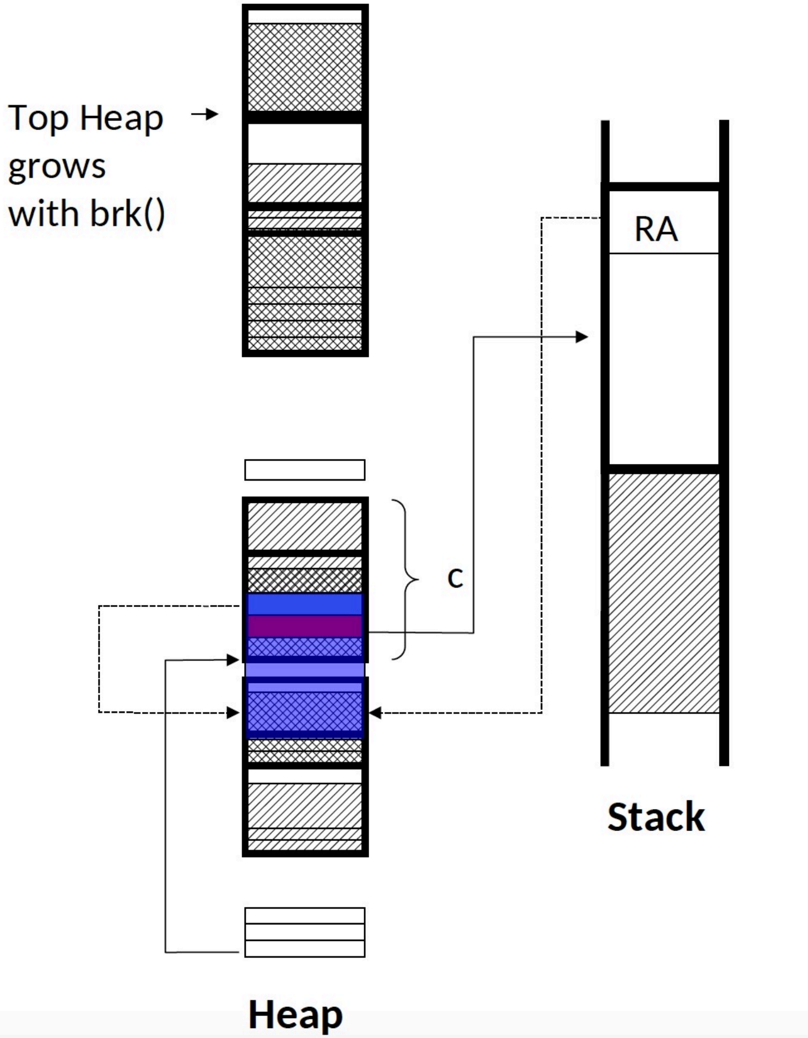
Green value is written 12 bytes above where red value points

A buffer overrun in d can overwrite the red and green values

. Make Green point to injected code

. Make Red point 12 bytes below a function return address

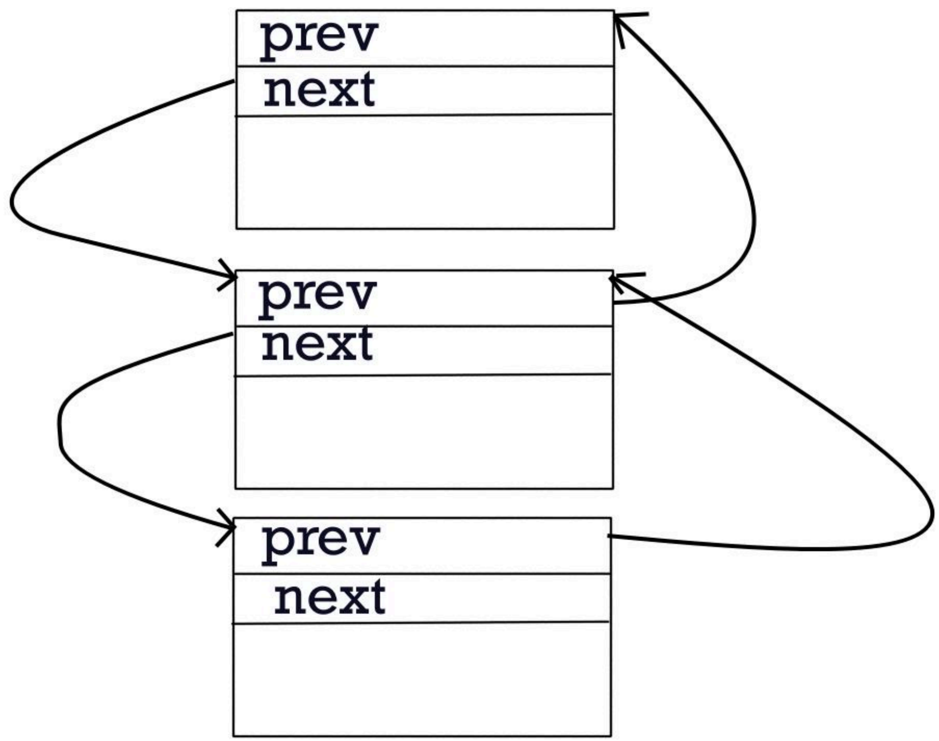
Exploiting a buffer overrun



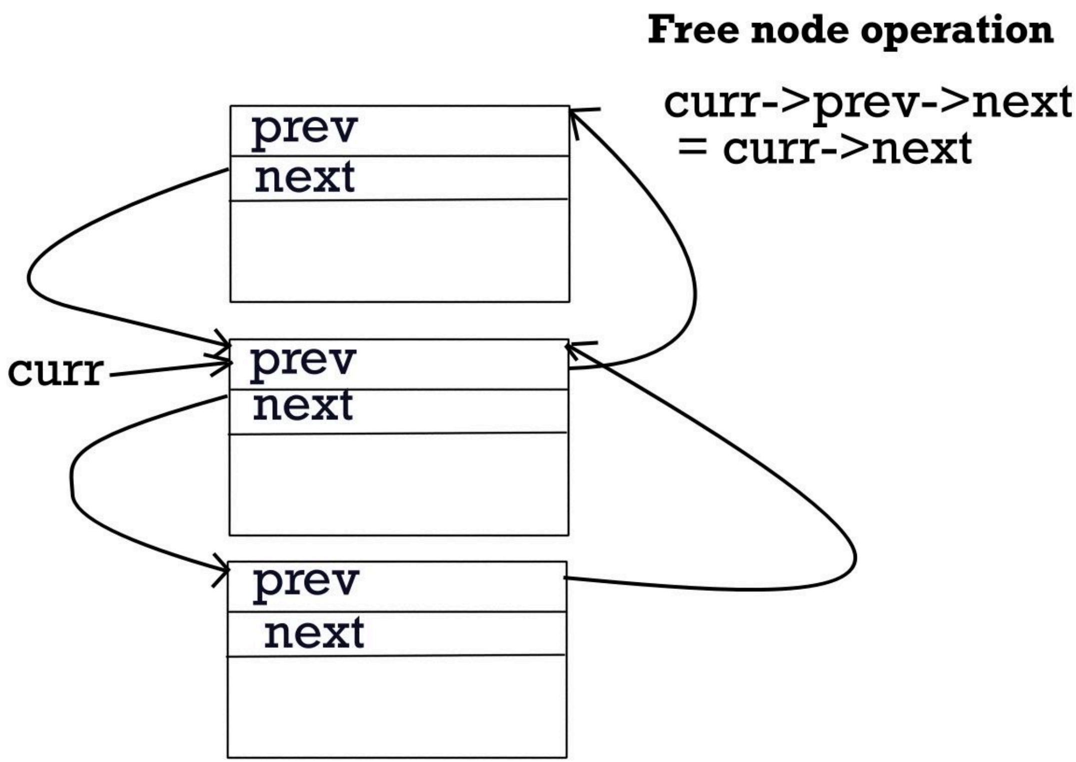
Green value is written 12 bytes above where red value points

Net result is that the return address points to the injected code

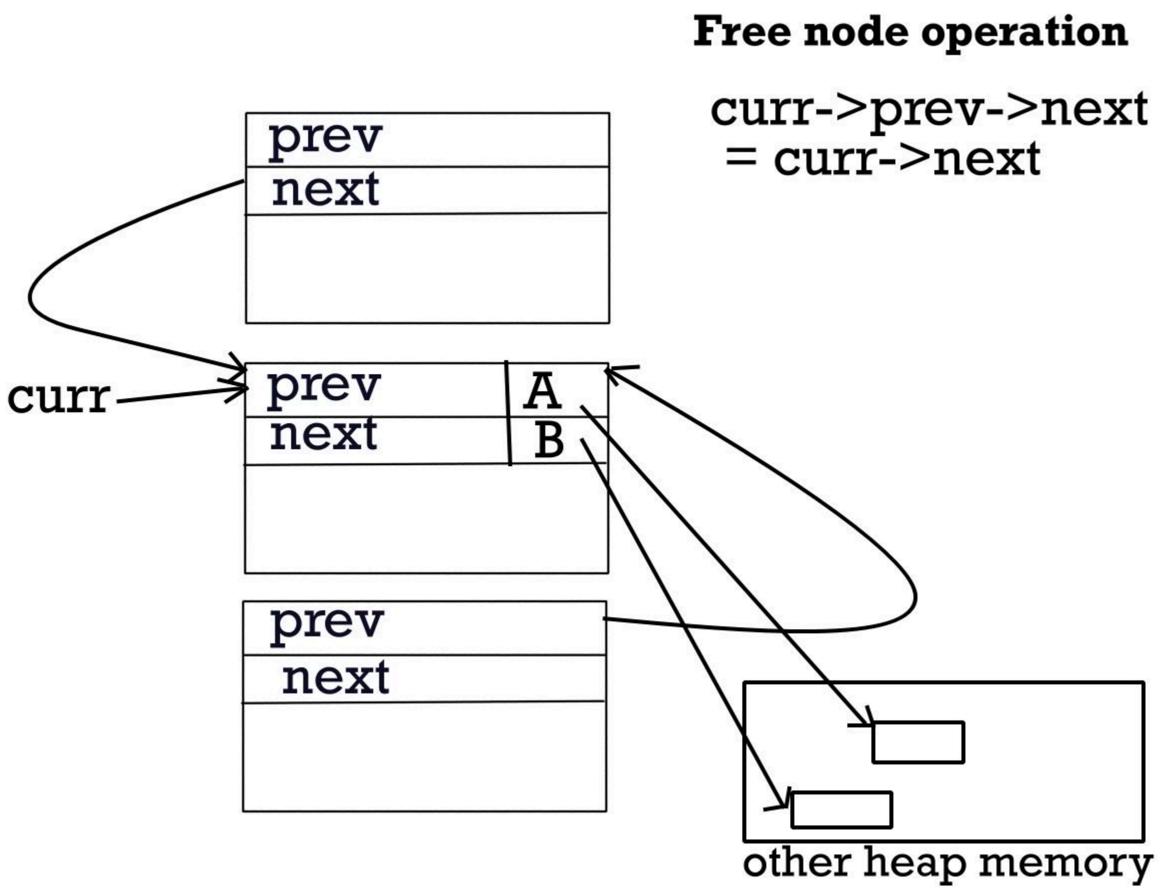
Heap Overflow of Mem Mgmt Metadata



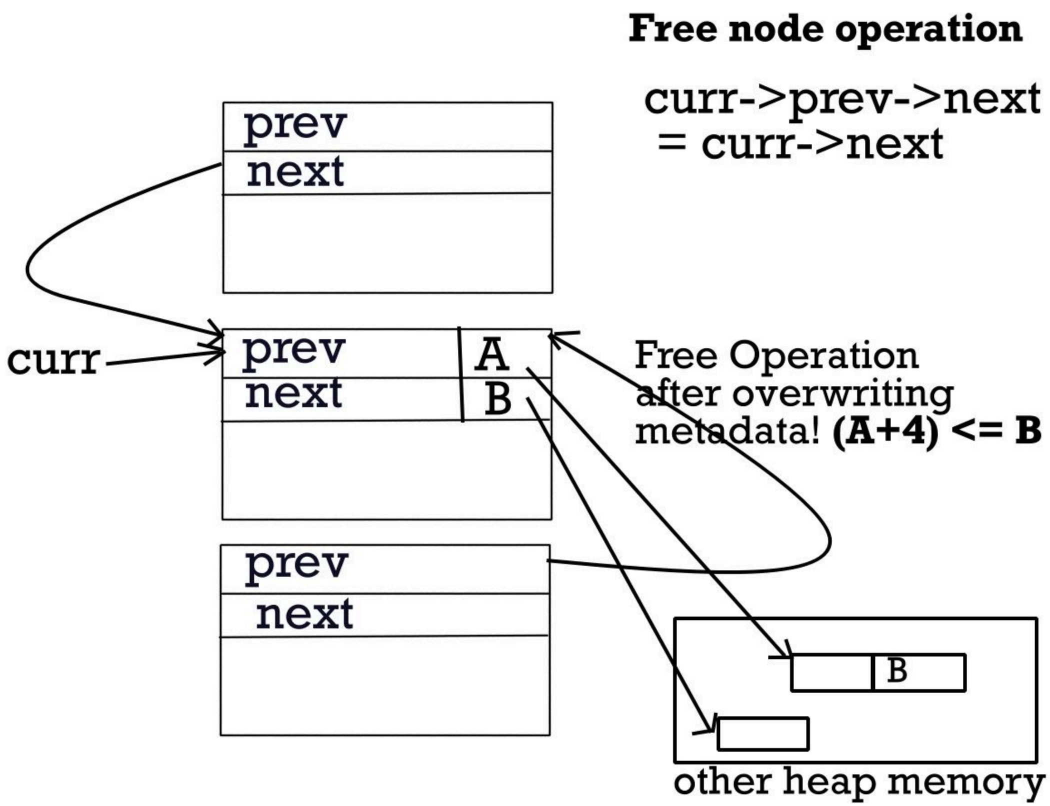
Heap Overflow of Mem Mgmt Metadata



Heap Overflow of Mem Mgmt Metadata



Heap Overflow of Mem Mgmt Metadata



Heap Overflows

- More generally, provides a primitive to write an arbitrary 32-bit value at an arbitrary location
- Possible targets
  - Function pointers
    - Return address on stack
      - Canaries don't help, but second RA copy will detect attack
    - Global Offset Table (GOT)
    - Function pointers in static memory
  - Data pointers
    - Names of programs executed or files opened
    - Application-specific data, e.g., "is_authenticated" flag in a login-like program

Heap Overflow Defenses

- Heap canaries
  - "magic numbers" between data and header
- Separation of metadata from data
  - In general, separating control data from program data is a good idea
    - Helps prevent data corruption attacks from altering the control flow of programs
  - Can be applied on the stack as well
    - "Safe stack" holds control-data
      - "safe" data (e.g., local integer-valued variables) can also be located there as they cannot be involved in memory errors
    - All other data moved to a second stack

Format-string Attacks

- Exploits code of the form
  - Read variables from untrusted source
  - printf(s)
- Printf usually reads memory, so how can it be used for memory corruption?
  - "%n" primitive allows for a memory write
  - Writes the number of characters printed so far (character count)
  - Many implementations (Linux, Windows) allow just the least significant byte of the number of character count
    - don't have to print large number of characters to write arbitrary 32-bit values --- perform 4 separate writes of the LSB of character count
    - Use field-width specifications to control character count
- Formatguard: pass in actual number of parameters so the callee can only dereference that many parameters
  - Not adopted in practice due to compatibility issues

Integer Overflows

- There are multiple forms
  - Assignment between variables of different width
    - Assign 32-bit value to 16-bit variable
  - Assignment between variables of different signs
    - Assign an unsigned variable to a signed variable or vice-versa
  - Arithmetic overflows
    - i = j+k
    - i = 4*j
    - Note that i may become smaller than j even if j > 0
- Exploitation
  - Allocate less memory than needed, leading to a heap overflow
    - One of the common forms of file-format attacks
  - "Escape" bounds checks
    - If (i < sizeof(buf)) memcpy(buf, src, i);
- For more info
  - http://www.phrack.org/archives/issues/60/10.txt

Memory Errors

- Although other attack types have emerged, memory errors continue to be the dominant threat
  - Behind most "critical updates" from Microsoft and other vendors
  - Mechanism of choice in "mass-market" attacks, including worms
  - Evolved to target client (web browsers, email-handlers, wordprocessors, document/image viewers, media players, ...) rather than server applications (e.g., web browsers)
- A memory error occurs when an object accessed using a pointer expression is different from the one intended
  - Spatial error : Examples
    - Out-of-bounds access due to pointer arithmetic errors
    - Access using a corrupted pointer
    - Uninitialized pointer access
  - Temporal error: access to objects that have been freed (and possibly reallocated)
  - Example: dangling pointer errors
  - applicable to stack and heap allocated data

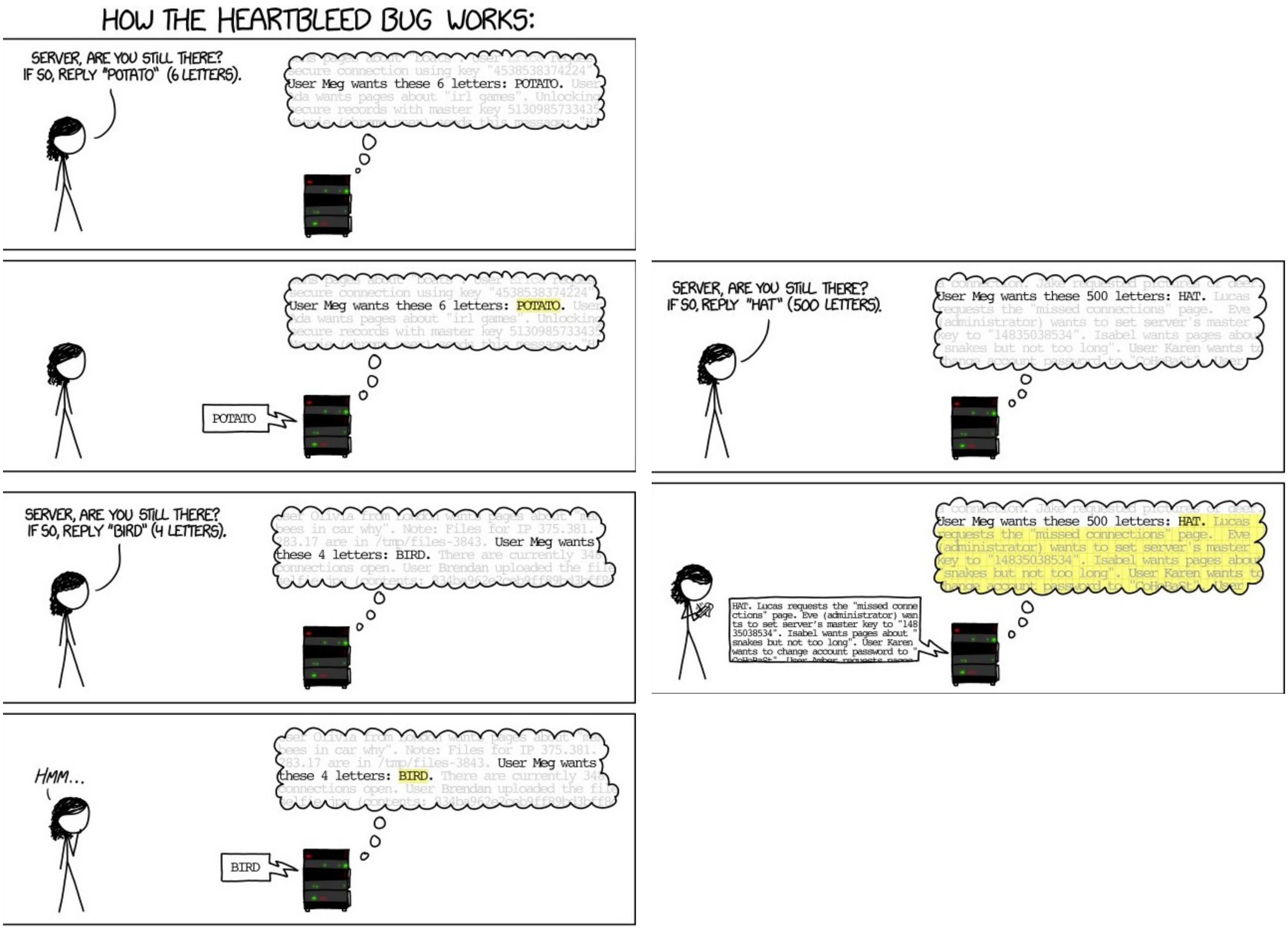
Memory Errors in C

- Spatial errors: out-of-bounds subscript or pointer
  - char *p = malloc(10); *(p+15);
- Temporal errors: pointer target no longer valid
  - Unintialized pointer
  - Dangling pointer
    - free(p); q = malloc(...); *p;
    - Note: target may be reallocated!
- Hard to debug, especially temporal errors
  - Unpredictable delay, unpredictable effect
    - Reallocated pointer errors are the worst kind
  - "Defensive programming" leads to memory leaks

Use of Memory Errors in Attacks

- Temporal errors
  - Not as frequently targeted as spatial errors, but are becoming more common ("double free," "use-after-free")
- Spatial errors
  - Pointer corruption is most popular
  - Out-of-bounds errors are most commonly used to corrupt pointers
    - But some attacks rely on just reads without necessarily corrupting existing data, e.g., heartbleed SSL vulnerability
- Typically, multiple errors (2-3) are used in an attack
  - Stack-smashing relies on out-of-bounds write, plus the use of a corrupted pointer as return address
  - Heap overflow relies on out-of-bounds write, use of corrupted pointer as target of write, and then the use of a corrupted pointer as branch target.

Overwrites aren't the only problem…


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

High-level Overview of Memory Error Defenses

- Block memory errors
  - Bounds-checking (mainly focused on spatial error)
    - Bounds-checking C and CRED, Valgrind memcheck, ...
    - Blocking all memory errors (including temporal)
- Disrupt exploits
  - Identify mechanisms used for exploit, block them
    - Disrupt mechanism used for corruption
      - Protect attractive targets against common ways to corrupt them ("guarding" solutions)
    - Disrupt mechanism used for take-over
      - Disrupt ways in which the victim program uses corrupted data
      - **Randomization-based defenses**
    - Disrupt payload delivery mechanism
      - DEP, CFI

A. Disrupting Memory Error Exploits

1. Disrupting mechanisms used for corruption

- Stackguard and related solutions
  - Protect RA/saved BP; with ProPolice, some local variables as well
- Magic cookies and safe linking on heaps
- Attacks on GOT
  - GOT contains function pointers used to call library functions
    - Compiler generates a stub for each library function in a code section called PLT (program linkage table)
    - Stub code for a function f performs an indirect jump using the address stored in the GOT corresponding to f.
  - Defense: hide GOT
    - Not very effective: injected code can search and locate it!
- Common problem for this approach: incomplete
  - Not all targets can be protected
  - Incomplete even for protected targets: some corruption techniques still succeed, e.g., corrupting RA without disturbing canary.

2. Disrupting payload delivery mechanisms

- Prevent control transfer to/execution of injected code
  - Most OSes enforce W ⊕ X (aka NX or DEP)
    - prevents writable memory from being executable, so can't execute injected code
  - Attackers get around this by reusing existing code
    - return-to-libc: return to the beginning of existing functions
      - Instead of having injected code spawning a shell, simply "return" to the execle function in libc
      - If it is a stack-smash, attacker controls the contents of the stack at this point, so they can control the arguments to execle
    - By constructing multiple frames on the stack, it is possible to chain together multiple fragments of existing code
      - ROP (return-oriented programming) takes this to the extreme
        - Chains together many small fragments of existing code ("gadgets")
        - Each gadget can be thought of as an "instruction" for a "virtual machine"
        - For sufficiently complex binaries, sufficient number and variety of gadgets are available to support

        Turing-complete computation
      - Most exploits today rely on ROP, due to widespread deployment of W ⊕ X
        - Goal of ROP payload is to invoke mprotect system call to disable W ⊕ X.

2. Disrupting payload delivery mechanisms

- Control-flow integrity (CFI) is another (partial) defense that limits attacker's freedom in terms of control transfer target
  - Can defeat most injected code and ROP attacks, but is not fool-proof
    - skilled attackers may be able to craft attacks that operate despite CFI

3. Disrupting take-over mechanism

- Key issue for an attacker:
  - using attacker-controlled inputs, induce errors with predictable effects
- Approach: exploit software bugs to overwrite critical data, and the behavior of existing code that uses this data
  - Relative address attacks (RA)
    - Example: copying data from input into a program buffer without proper range checks
  - Absolute address attacks (AA)
    - Example: store input into an array element whose location is calculated from input.
      - Even if the program performs an upper bound check, this may not have the intended effect due to integer overflows
  - RA+AA attacks: use RA attack to corrupt a pointer p, wait for program to perform an operation using *p
    - Stack-smashing, heap overflows, …

Disrupting take-over: Diversity Based Defenses

- Software bugs are difficult to detect or fix
  - Question: Can we make them harder to exploit?
- Benign Diversity
  - Preserve functional behavior
    - On benign inputs, diversified program behaves exactly like the original program
  - Randomize attack behavior
    - On inputs that exercise a bug, diversified program behaves differently from the original

Automated Introduction of Diversity

- Use transformations that preserve program semantics
- Challenge: how to capture intended program semantics?
  - Relying on manual specifications isn't practical
- Solution: Instead of focusing on program-specific semantics, rely on programming language semantics
  - Randomize aspects of program implementation that aren't specified in the programming language
    - Benefit: programmers don't have to specify any thing

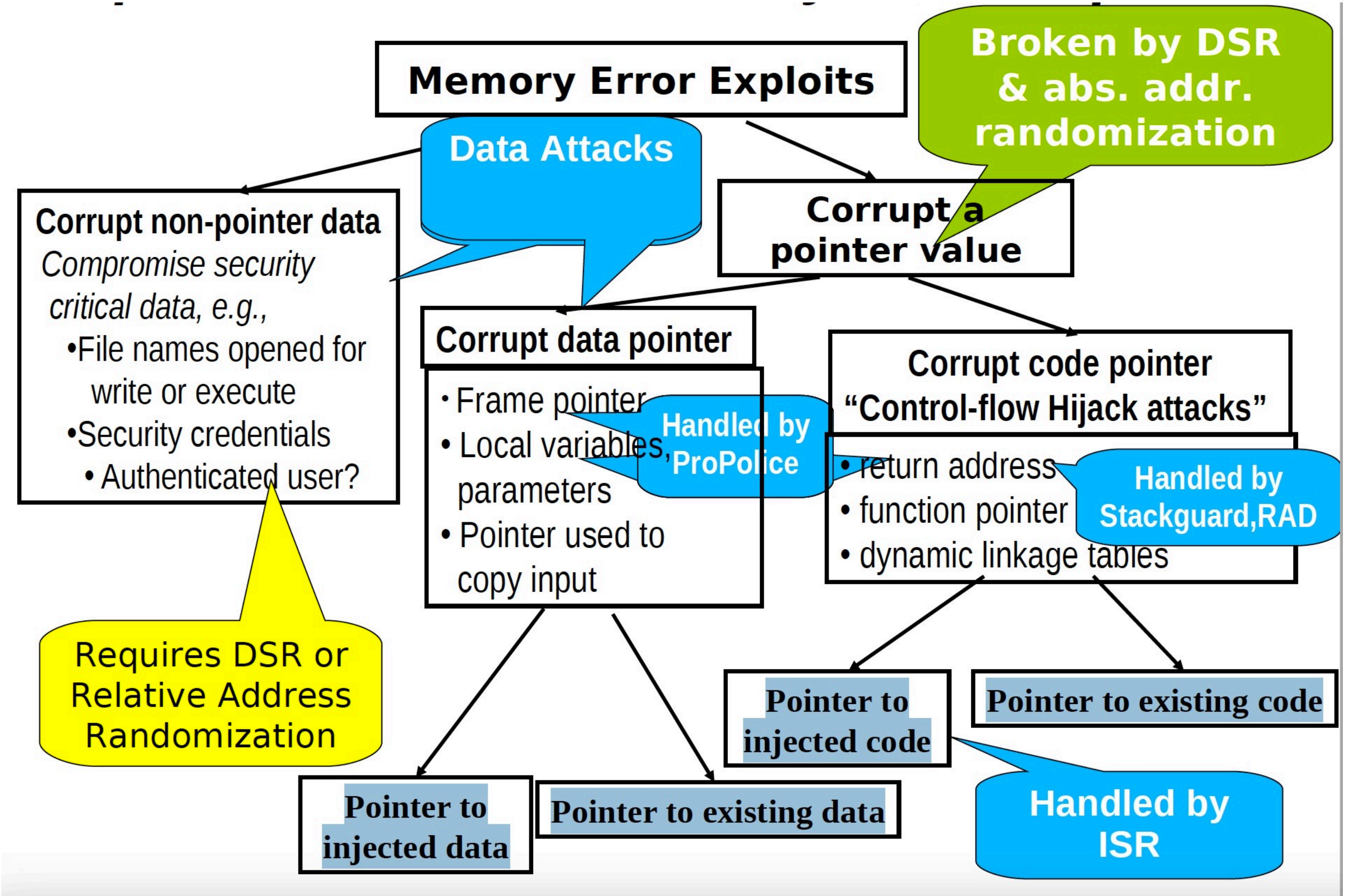
Automated Introduction of Diversity

- Examples
  - Address Space Randomization (ASR)
    - Randomize memory locations of code or data objects
    - Invalid and out-of-bounds pointer dereferences access unpredictable objects
  - Data Space Randomization (DSR)
    - Randomize low-level representation of data objects
    - Invalid copy or overwrite operations result in unpredictable data values
  - Instruction Set Randomization (ISR)
    - Randomize interpretation of low-level code
    - $W \oplus X$ has essentially the same effect, so ISR is not that useful any more

How randomization disrupts take-over

- Without randomization, memory errors corrupt process memory in a predictable way
  - Attacker knows what data is corrupted, e.g., return address on the stack
    - Relative address randomization (RAR) takes away this predictability
  - Attacker knows the correct value to be used for corruption, e.g., the location of injected code (in a buffer that contains data read from attacker)
    - Absolute address randomization (AAR) takes away this predictability for pointer-valued data
    - DSR takes away this predictability for all data

Space of Possible Memory Error Exploits



First Generation ASR: Absolute Address Randomization (ASLR)

- Invented by PaX project and Our Lab at SBU
- Randomizes base address of data (stack, heap, static memory) and code (libraries and executable) regions
- Implemented on many flavors of UNIX & Windows
  - UNIX implementations usually provide 20+ bits of randomness, 16 bits for Windows [about 20 lines of code in kernel]
- Implemented on all mainstream OS distributions
  - Linux, OpenBSD, Windows, Android, iOS, ...
- Limitations
  - Incomplete implementations (e.g., executables or some libraries left unrandomized) --- but this is becoming rare these days.
  - Brute-force attacks
  - Information leakage attacks
  - Relative address attacks
    - Non-pointer data attacks, partial pointer overwrites

Second Generation ASR: Relative Address Randomization

- Randomize distance between objects (code or data)
- [Bhatkar et al] use code transformation to permute the relative order of objects in memory
  - Static variables
  - “Unsafe” local variables
    - Safe local variables moved to a “safe” stack (no overwrites possible)
    - Safe stack option is now available on LLVM compiler
  - Heap allocations
  - Functions
  - Introduce gaps between objects
    - Some gaps may be made inaccessible
- **Active current research: efficient RAR of code objects**

Benefits of RAR

- Defeats the overwrite step, as well the step that uses the overwritten pointer value
  - Defeats format-string and integer overflow attacks
  - Stack-smashing attacks fail deterministically (due to safe stack)
- Higher entropy
  - Up to 28 bits on 32-bit address space
  - Knowing the location of one object does not tell you much about the locations of other objects
    - information leakage attacks become difficult
    - heap overflows become more difficult since you need to make two independent guesses

Data Space Randomization

DSR Technique

- Basic idea: Randomize data representation
  - Xor each data object with a distinct random mask
  - Effect of data corruption becomes non-deterministic, e.g.,
    - Use out-of-bounds access on array a to corrupt variable x with value v
      - Actual value written: mask(a) xor v
      - When x is read, this value is interpreted as mask(x) □ (mask(a) xor v)
        - Which is different from v as long as the masks for x and a differ.
- Benefits
  - Unlike AAR, protects all data, not just pointers
  - Effective against relative address as well as absolute address attacks
  - Large entropy
    - 32-bits of randomization for integers
    - Masks for different variables can be independent
      - resists information leak attacks
  - Can address intra-structure overflows
    - Not even addressed by full memory error detection techniques

DSR Transformation Approach

- For each variable v, introduce another variable m_v for storing its mask
- Randomize values assigned to variables (LHS)
  - Example: `x = 5 ==> x = 5; x = x ^ m_x;`
- Derandomize used variables (RHS)
  - Example: `(x + y) ((x ^ m_x) + (y ^ m_y))`
- Key problem: aliasing
  - `int *x = &y`
  - A value may be assigned to y and dereferenced using `*x`
    - Both expressions should yield the same value
      - Need to ensure that possibly aliased objects should use the same randomization mask
- Note
  - In `x = y`, it is not necessary to assign same mask to x and y

Summary of Automated Diversity

- Transformations that respect programming language semantics are good candidates for automated diversity
  - But they are typically good for addressing only low-level implementation errors. (We have discussed them only in the context of a specific low-level error, namely, memory corruption.)
- Automated diversity has been particularly successful in the area of memory error exploit prevention
  - First generation of randomization-based defenses focused on absolute address based attacks
    - Absolute-address randomization
    - Practical technique with low impact on systems, and hence begun to be deployed widely
  - Second generation defenses provide protection from relative-address dependent attacks
    - Relative address randomization and data-space randomization
    - Performance and compatibility (for DSR) limit widespread deployment

State of Exploit defenses and New attacks

- Most OSes now implement
  - ProPolice like defenses, plus SEH protection (Microsoft)
  - ASLR
  - DEP/NX (prevent injected code execution)

State of Exploit defenses and New attacks

- Recent attacks
  - Exploit incomplete defenses, or use Heapspray for control-flow hijack
    - No ASLR on most executables on Linux, some EXE, DLLs on MS
    - Some libraries don't enable stack protection, or it is incomplete
    - Heapspray: brute-force attack in the space domain
      - Exploits untrusted code in safe languages (Javascript, Java, Flash,...)
      - Code allocates almost all of memory, fills with exploit code
      - Jump to random location: with high probability, it will contain exploit code
  - Return-oriented programming (ROP) to overcome DEP
  - Rely increasingly on information leak attacks to overcome uncertainty due to ASLR, frequent software updates, and so on
    - Just-in-time-ROP: use information leak vulnerability to scan code at runtime to identify ROP gadgets

B. Preventing Memory Errors

Memory Errors in C

- Spatial errors: out-of-bounds subscript or pointer
  - char *p = malloc(10); *(p+15);
- Temporal errors: pointer target no longer valid
  - Unintialized pointer
  - Dangling pointer
    - free(p); q = malloc(…); *p;
    - Note: target may be reallocated!
- Hard to debug, especially temporal errors
  - Unpredictable delay, unpredictable effect
    - Reallocated pointer errors are the worst kind
  - "Defensive programming" leads to memory leaks

Issues and Constraints

- Backward compatibility with existing C-code
  - Casts, unions, address arithmetic
  - Conversion between integers and pointers
- Compatibility with previously compiled libraries
  - Can't expect to rebuild the entire system
  - Source code access can be problematic for some libs
- Temporal Vs Spatial Errors
  - Detecting reallocated storage
  - Important, since such errors get detected very late, and it is extremely hard to track them down
- Use of garbage collection

Why Not Garbage Collection?

- Masks temporal errors
  - Problematic if the intent is to use memory error-checking only during the testing phase
- Unpredictable overheads
  - Problematic for systems with real-time or stringent performance constraints
- GCs can make mistakes due to free conversion between integers and pointers
  - Fail to collect inaccessible memory
  - Collect memory that should not be collected
  - Problematic for code that relies heavily on such conversions, e.g, OS Kernel

Approaches for Preventing Memory Errors

- Introduce inter-object gaps, detect access to them (Red zones)
  - Detect subclass of spatial errors that involve accessing buffers just past their end
  - Purify, Light-weight bounds checking [Hasabnis et al], Address Sanitizer [Serebryany et al]
- Detect crossing of object boundaries due to pointer arithmetic
  - Detects spatial errors
  - Backwards-compatible bounds checker [Jones and Kelly 97]
  - Further compatibility improvements achieved by CRED [Ruwase et al]
  - Speed improvements: Baggy [Akritidis et al], Paricheck [Younan et al]

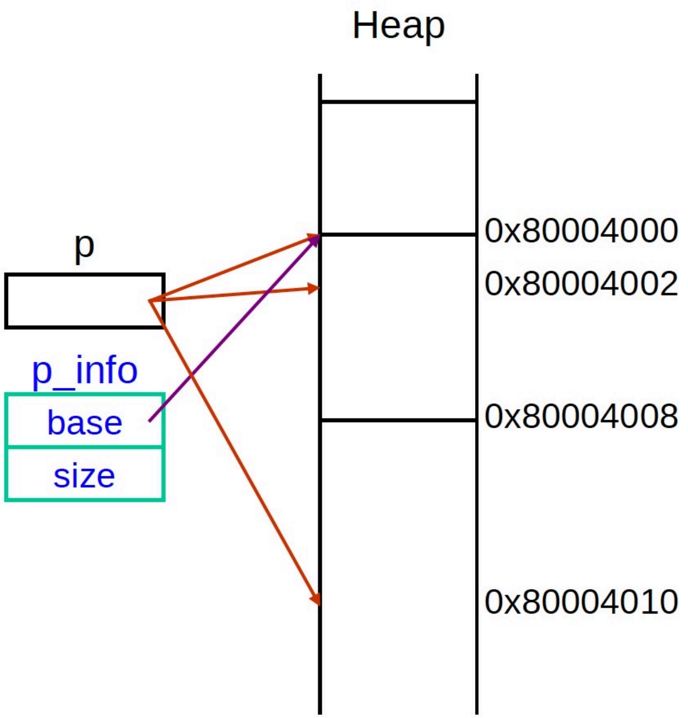
Approaches for Preventing Memory Errors

- Runtime metadata maintenance techniques
  - Temporal errors: pool-based allocation [Dhurjati et al], Cling [Akritidis et al]
  - Spatial + temporal errors: CMemSafe [Xu et al], SoftBounds [Nagarakatte et al]
  - Targeted approaches: Code pointer integrity [Kuznetsov et al], protects subset of pointers needed to guarantee the integrity of all code pointers.

CMemSafe: Detecting Spatial Errors Using Metadata

```
Spatial Check:
(p >= p_info.base &&
 p < p_info.base+p_info.size)?
```

```
char * p;

p = malloc(8);

p += 2;

*p;    /* OK */

p += 14;

*p;    /* error */
```

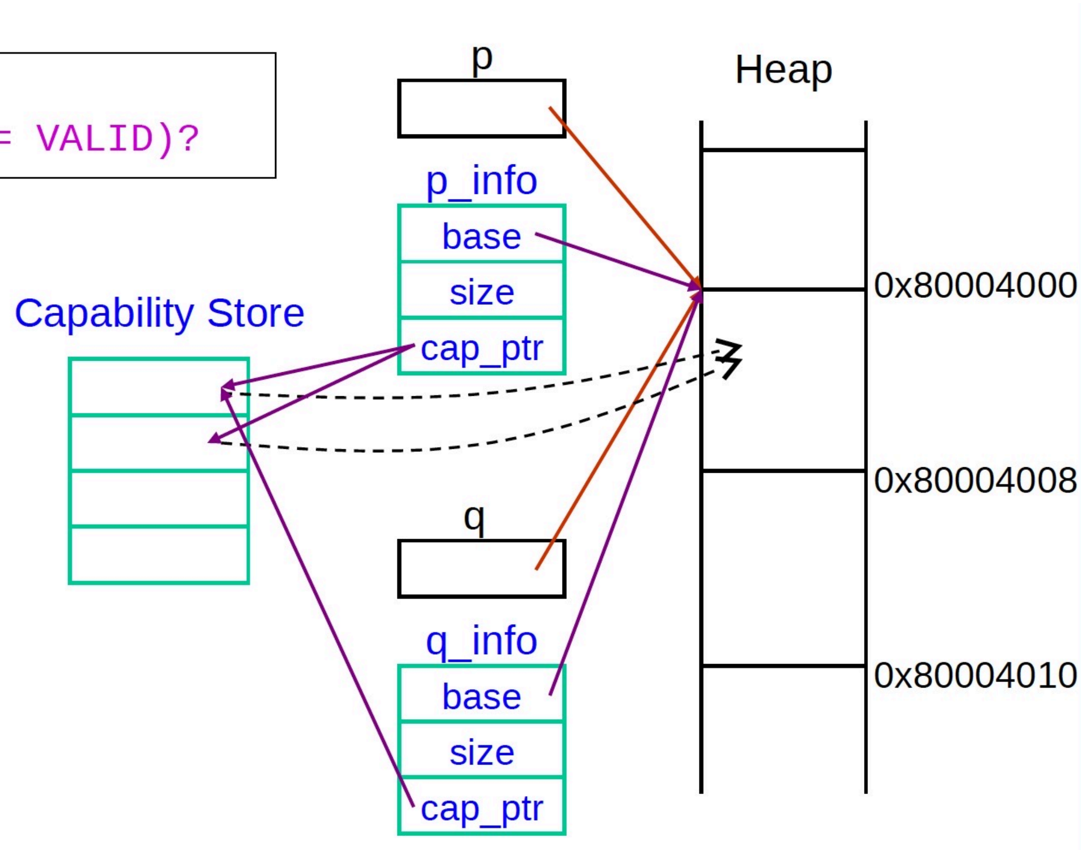


base, size: base address and allocated size of the block

CmemSafe: Detecting Temporal Errors

```
Temporal Check:
(*q_info.cap_ptr == VALID)?
```

```
char * p, *q;
p = malloc(8);
q = p;
*q;    /* OK */
free(p);
*q;    /* error */
p = malloc(16);
*q;    /* error */
```



cap_ptr: pointer to unique capability associated with block

Detect erroneous accesses to freed or reallocated memory

Credits

- Slides on Stack layout, ROP and heap overflows: courtesy Nick Nikiforakis

Questions