CSCI 4621/5621 Intro to CyberSecurity

03: Exploits & Privilege Escalation

Vassil Roussev

vassil@cs.uno.edu

READING: [Oorschot [ch6]]

GOAL

- Review essential technical approaches used by attackers
  - *i.e., understand the common attack vectors*
  - ➔ *devise technical defenses to thwart them*
- This is an inherently *reactive* approach
  - *puts the attacker one step ahead*
- Most security research:
  - *try to find the vulnerabilities **before** the attackers*
- Ideally, we should be producing software with no vulnerabilities
  - *but we don't quite know how*
  - *the ever-growing complexity of software is **not** helping*

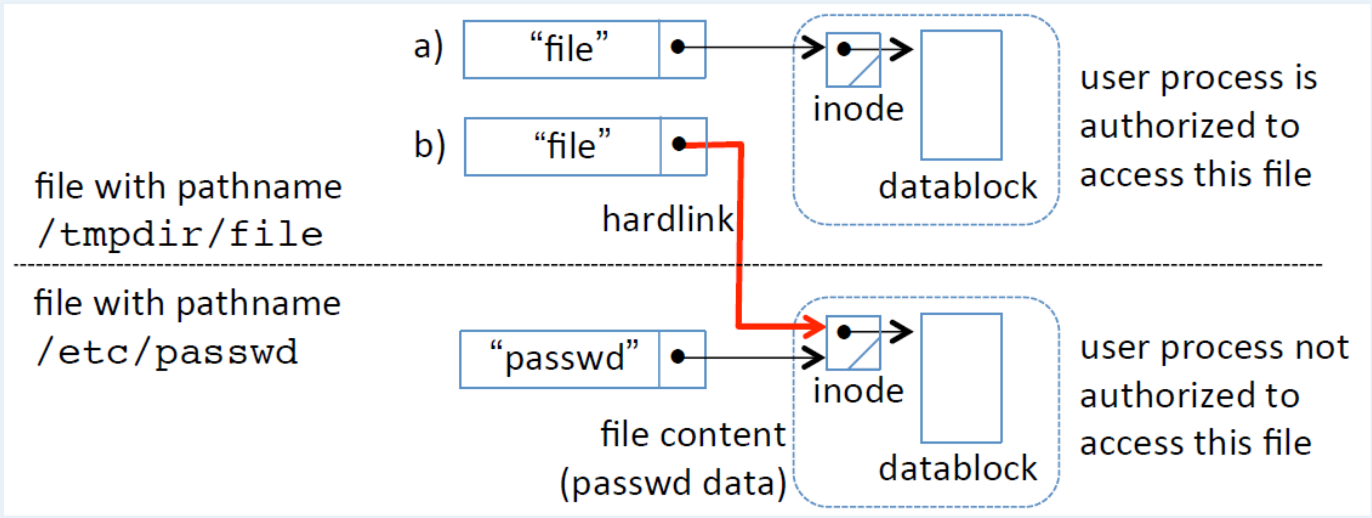
RACE CONDITIONS/HAZARDS

RACE CONDITION/HAZARD

- In a concurrent system:
  - *the outcome/correctness depends on the order of task execution*
- May have security implications
  - *by breaking assumptions built into the security mechanisms*
  - *an attacker may be in a position to influence the outcome to make an attack more likely to succeed*

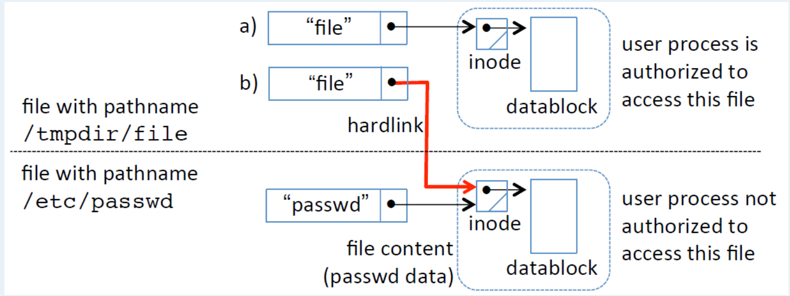
TIME-OF-CHECK TO TIME-OF-USE [**TOCTOU**]

- Consider AC check for file access:
  - t_1 → *time of check*
  - t_2 → *time of use*
  - $t_1 < t_2$
- Potential problem:
  - *if check is successful, the result is used "immediately after"*
  - *however, the environment may change in between due to concurrent processing*



[CREDIT: Oorschot]

Ex: Privilege Escalation via TOCTOU



- setuid
  - *root-owned*
- Common implementation

```
if(access("file",PERMS REQUESTED)==0) then
    filedescr = open("file", PERMS)  /* now proceed to read or write */
```

- Problem? – attacker may alter the binding b/w filename & content

```
unlink("file")                /* delete name from filesystem */
link("/etc/passwd", "file")   /* new file entry links to passwd */
```

Mitigating races

- Underlying problem
  - *non-atomic execution of the lines of code*
- One solution
  - *use file descriptors, instead of filename*
    - those are immutable

Ex: file squatting

- /tmp → 1777 per missions (sticky bit)
- gcc
  - main.c → xyzw.i → xyzw.s → xyzw.o
- Attacker
  - *waits for the appearance of a* `xyzw.i` *file*
  - *creates a symlink to* `xyzw.o` *to a different file (before* `gcc`*)*
  - `gcc` *uses* `O_CREAT` *access and may overwrite the target file*
    - given sufficient permissions

INTEGER-BASED VULNERABILITIES & C

DEFINITION

- Exploitable code due to integer bugs
  - *i.e., errors related to the internal representation of ints*
- C/C++ are the main culprit
  - *a **LOT** of system software is written in C/C++*
- C provides a very thin layer of abstraction
  - *it aims to provide efficient access to the underlying hardware*
    - results may require low-level understanding of what the results of operations are
  - *it is considered "weakly-typed"*
    - i.e., any type can be cast/converted to any type

C CHAR

- three types
  - char, unsigned char, signed char
- char arithmetic
  - *implementation dependent conversion before arithmetic operations*
  - *either*
    - char → unsigned char
    - char → signed char
- C99
  - int8_t, int16_t, int32_t, int64_t

Data type	Bit length	Range	
		unsigned	signed
char	8	0..255	−128..127
short int	16	0..65535	−32768..32767
int	16 or 32	0..UINT_MAX	INT_MIN..INT_MAX
long int	32	$0..2^{32}-1$	$-2^{31}..2^{31}-1$
long long	64	$0..2^{64}-1$	$-2^{63}..2^{63}-1$

[CREDIT: Oorschot]

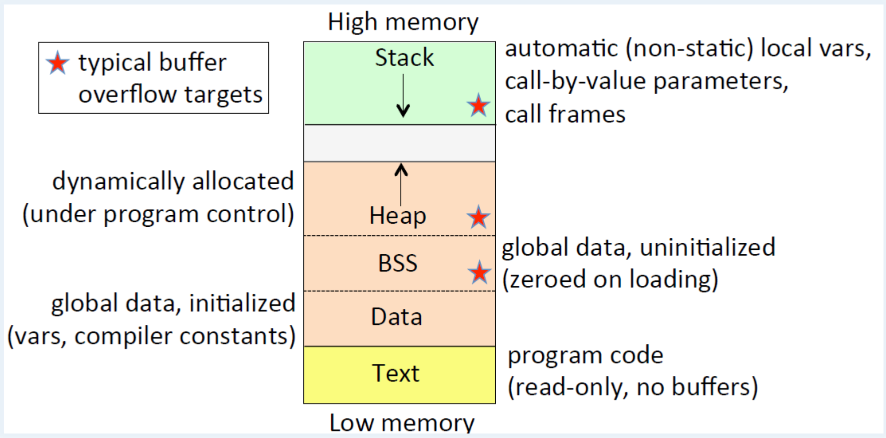
INTEGER PROBLEMS IN C

- Conversions
  - *C promotes* char *and* short *to* int → *result is different:*
    - unsigned: zero-extended
    - signed: sign-extended (sign bit propagated)
  - *down conversions truncate high order bits*
  - *same width* signed ⬄ unsigned
    - no change in bits, just interpretation
- Integer overflow
  - unsigned char x = 255
  - x++ == ??
    - result it 0 – overflow bit is ignored ➔ modular wrapping

Integer problems in C [2]

- Modular wrapping vs. undefined
  - **signed** char x = 127
  - x++ == ??
    - result is undefined (machine-dependent)
    - in practice, often wraps around to a negative value
- Underflow
  - **signed** char x = -128
  - x++ == ??
- Overflow/underflow is not C's concern
  - *no errors, exceptions, etc.*
  - *that can lead to logic problem in the code*
    - incrementing a positive number can yield a negative one!

EX: FROM OVERFLOW TO DATA ACCESS



[CREDIT: Oorschot]

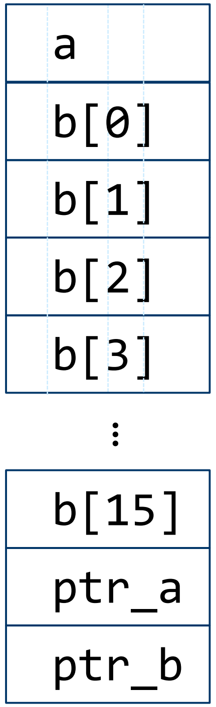
- Assume 16-bit machine
  - unsigned int width, height
  - *values are attacker controlled (e.g., user input)*
  - char *matrix = (char *) malloc(width*height)
- Consider
  - width = 235, height = 279
  - 235 * 279 = 65565 = 65536 + 29 = 216 + 29 = 0x1001D
  - (int) 0x1001D = 0x1D
  - malloc *only allocates 29 bytes (instead of 65565)*
  - *now read/write* matrix[i][j] *accesses reach something else in memory*
    - which might well be predictable

INTEGER PROBLEMS IN C—SUMMARY

Category	Description	Examples
integer overflow	value exceeds maximum representable in data type, e.g., >INT_MAX (signed) or >UINT_MAX	adding 1 to 16-bit UINT `0xFFFF` yields not `0x10000`, only low-order 16 bits `0x0000` multiplying 16-bit UINTs produces a 16-bit result in C, losing high-order 16 bits
integer underflow	value below minimum representable in data type, e.g., (unsigned) <0 or (signed) <INT_MIN	subtracting 1 from `signed char` `0x80` (−128) yields `0x7F` (+127), i.e., wraps subtracting 1 from 16-bit UINT `0x0000` (0) yields `0xFFFF` (+65535)
signedness mismatch (same-width integers)	signed value stored into unsigned (or vice versa)	assigning 16-bit SINT `0xFFFE` (−2) to 16-bit UINT will misinterpret value (+65534) assigning 8-bit UINT `0x80` (128) to 8-bit SINT changes interpreted value (−128)
narrowing loss	on assigning to narrower data type, truncation loses meaningful bits or causes sign corruption	assigning 32-bit SINT `0x0001ABCD` to 16-bit SINT loses non-zero top half `0x0001` assigning SINT `0x00008000` to 16-bit SINT gives representation error `0x8000` (-2^{15})
extension value change	sign extension of signed integer to wider unsigned (Beware: `short`, and also often `char`, are signed)	assigning `signed char` `0x80` to 32-bit UINT changes value to `0xFFFFFF80` assigning 16-bit SINT `0x8000` to 32-bit UINT changes value to `0xFFFF8000`

[CREDIT: Oorschot]

C POINTER ARITHMETIC



- Pointers contain addresses
  - » int a = 0, b[16]
  - » int *ptr_a = &a, *ptr_b = b+4   /* b[4] */
  - » *ptr_a = 1
- C allows addition/subtraction
  - » *arrays are indexed using a subscript operator, e.g.:*
    - b[i] evaluates ((b)+(i)) in pointer arithmetic
    - then dereferences the resulting address to extract a value denoted *(b+i) as “pointer and offset”.
    - (i) is computed w/ arithmetic conversion and promotion rules as above
      - negative results are allowed
  - » *offset is scaled depending on element type; e.g.,* sizeof(int)
- Q: a[2] == ?   b[16] == ?   ptr_b – ptr_a == ?

C POINTER ARITHMETIC—**CONSEQUENCES**

- Memory safety violations
  - *indexes **outside** the declared range*
- Smaller than needed dynamic memory allocation
  - malloc() *example earlier*
  - *can lead to **buffer overflows***
- Integer underflow → negative argument to malloc()
  - *request for a huge chunk of memory*
    - may trigger out-of-memory situation
    - or may not be satisfied ➔ return NULL
- Integer overflow
  - *may lead to excessive number of loop iterations*
  - *access to outside allocation*

INTEGER BUG MITIGATION

- ALU flags not directly accessible from C
  - » *need assembly instructions to check them*
  - » *gcc/clang offer options to generate code*
- Mostly, it is up to the developer
  - » *safety-check all parameters*
  - » *use safe libraries*

Bitstring	Unsigned	one's complement	two's complement	Notes
0000	0	represents same value as unsigned		leftmost bit 0 signals positive integer remaining bits specify magnitude
0001	1			
0010	2			
0011	3			
0100	4			
0101	5			
0110	6			
0111	7			
1000	8	−7	−8	leftmost bit is sign bit (1 if negative) one's complement has a redundant −0; two's complement has an extra value
1001	9	−6	−7	
1010	10	−5	−6	
1011	11	−4	−5	
1100	12	−3	−4	
1101	13	−2	−3	
1110	14	−1	−2	
1111	15	−0	−1	

[CREDIT: Oorschot]

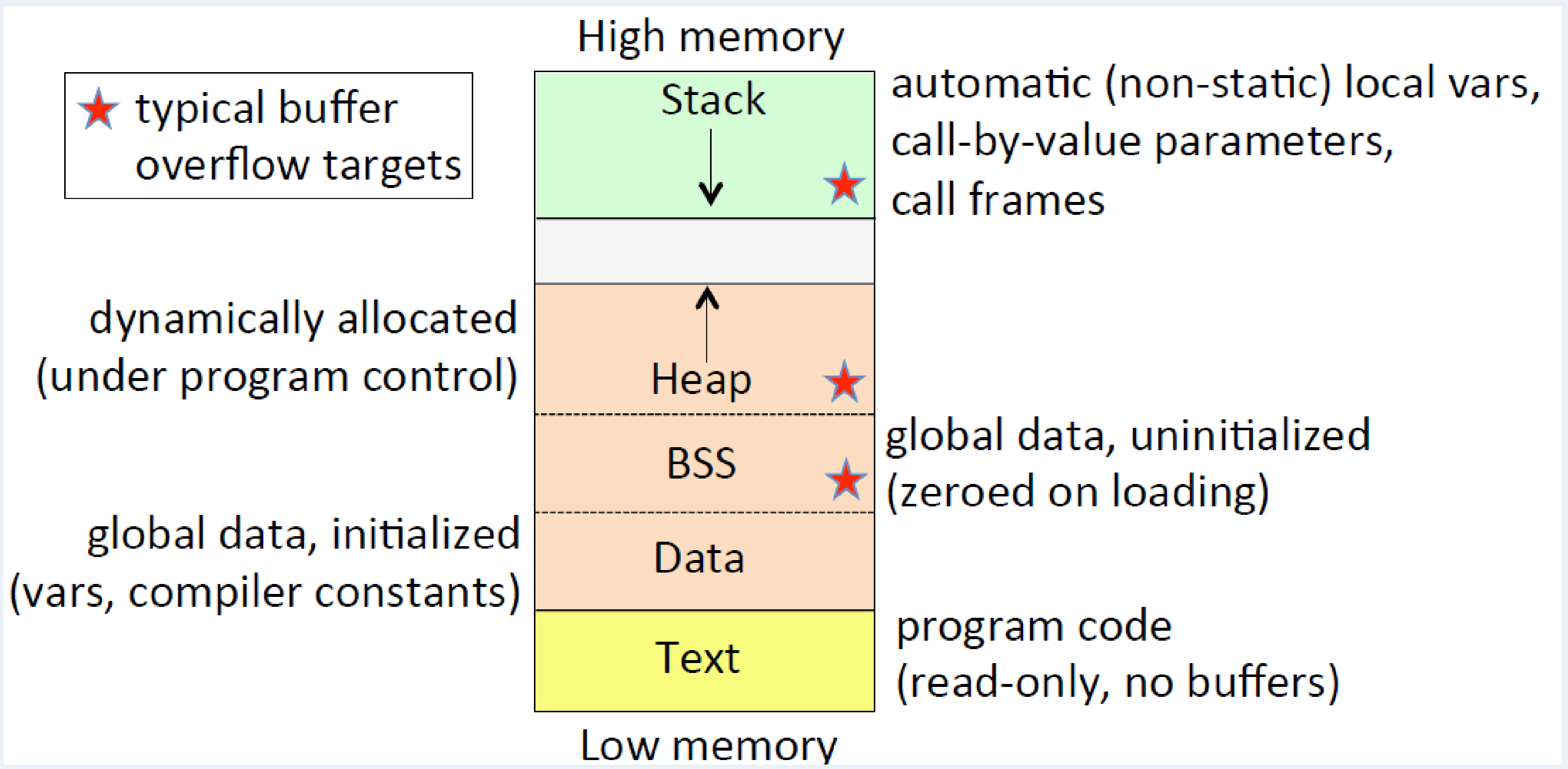
Dealing w/ pointer arithmetic concerns

- It is already deployed widely
  - *need to expect & deal with the consequences*
- It is an example of why we need *datatype safety*
  - *most modern languages are strongly-typed, e.g., Java*
    - cannot cast anything-to-anything
    - array bounds checking
- Large number of security concerns due to lack of memory safety
  - *any part of the code can access allocated stack/heap*
    - large attack surface for poorly written code

STACK-BASED OVERFLOWS

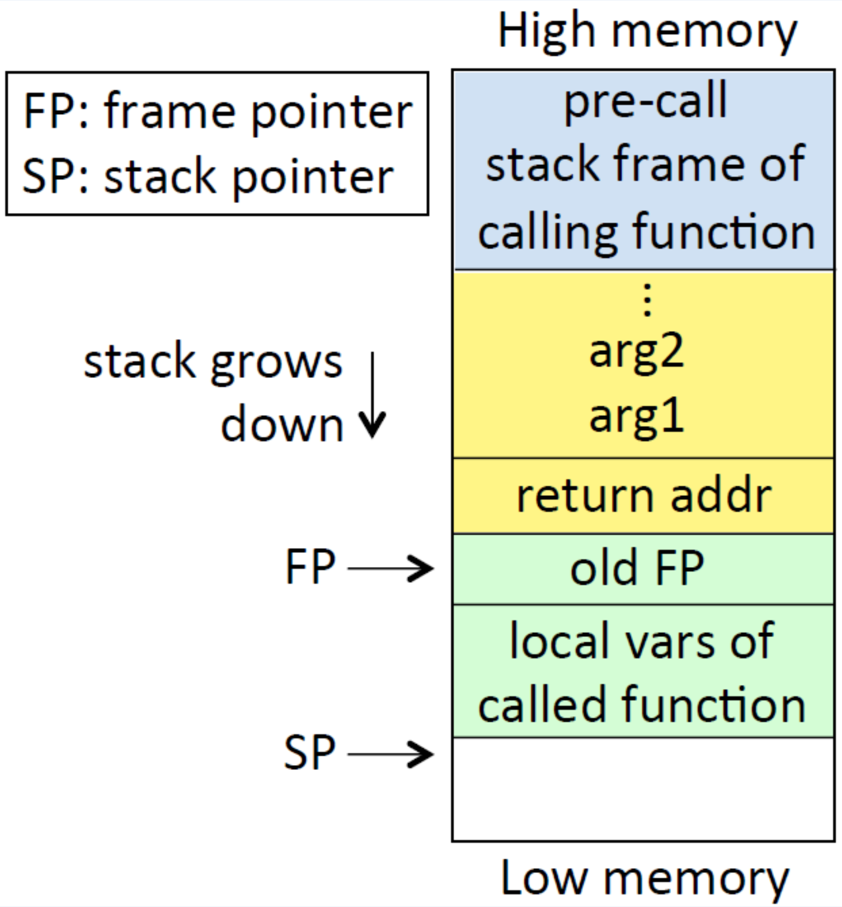
stack overflows

MEMORY LAYOUT (REVIEW)



[CREDIT: Oorschot]

Function calls (x86, review)



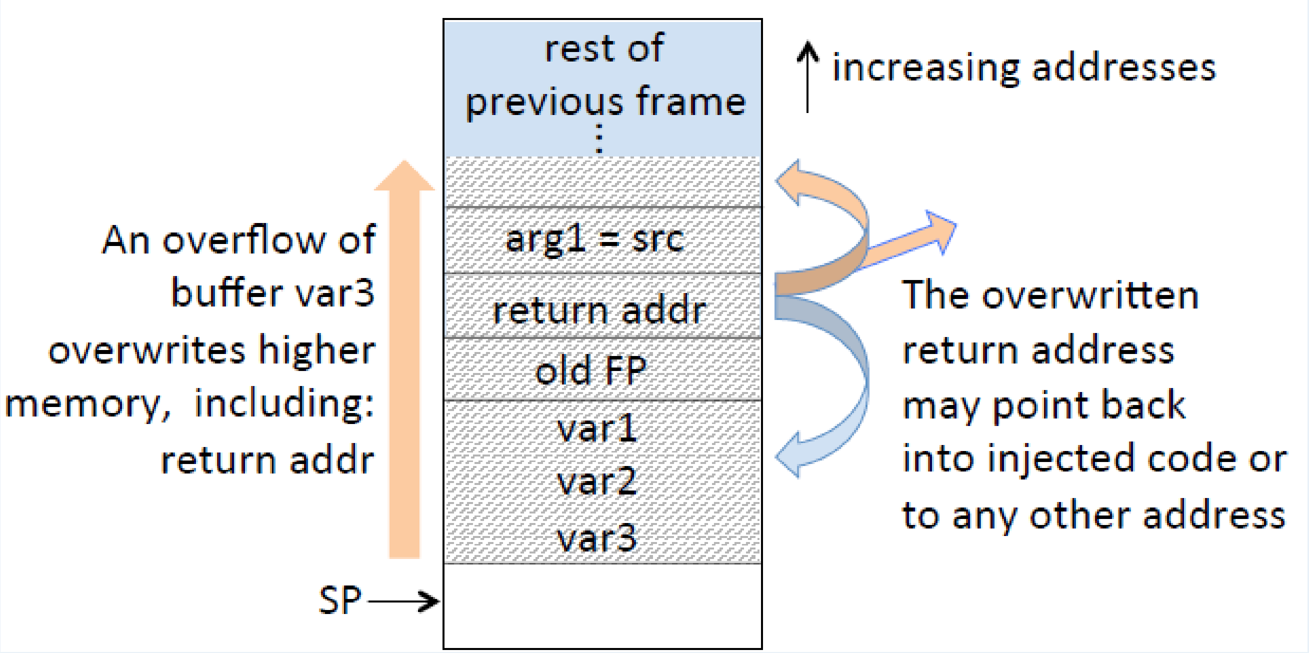
1. calling function pushes args onto stack
2. "call" opcode pushes Instruction Pointer (IP) as return address, then sets IP to begin executing code in called function
3. called function pushes FP for later recovery
4. FP ← SP (so FP points to old FP), now FP+k = args, FP-k = local vars
5. decrement SP, 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

[CREDIT: Oorschot]

EX: BUFFER OVERFLOW

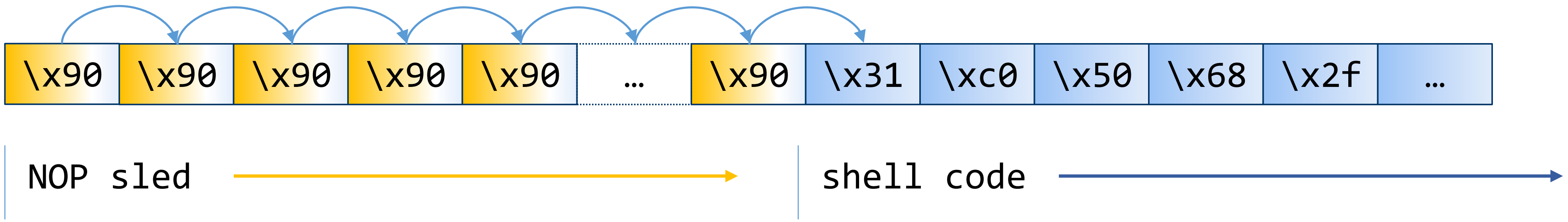
```
void myfunction(char *src)    /* src is a ptr to a char string */
{  int var1, var2;            /* 1 stack word used per integer */
   char var3[4];              /* also 1 word for 4-byte buffer */

   strcpy(var3, src);         /* template:  strcpy(dst, src)   */
}
```



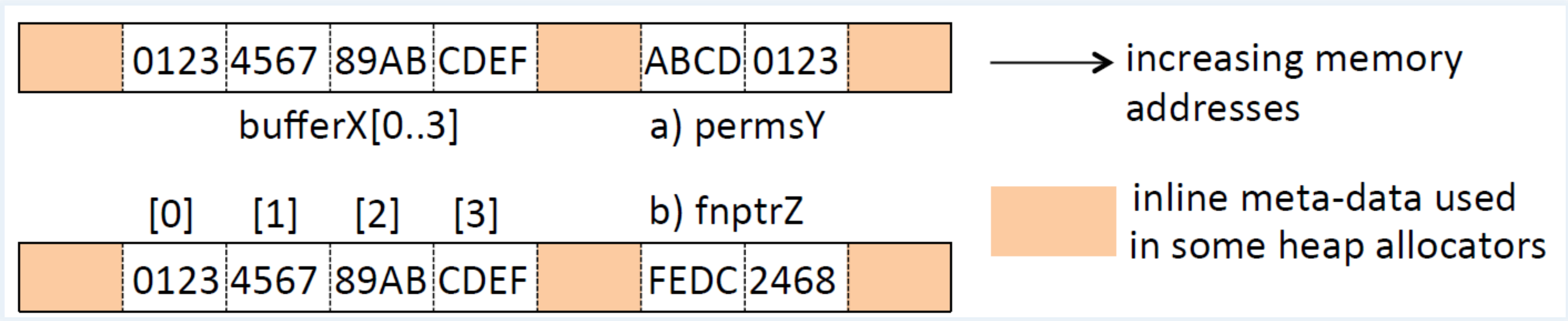
NO-OP SLED

- A sequence of dummy instructions
  - *has not effect on the state of the CPU*
  - *could be NOP instruction*
    - too visible
  - *or, it could be meaningless computations (e.g., adding zero to registers)*
- The point is to provide a *range* of addresses for the hijacked return addr



HEAP-BASED BUFFER OVERFLOWS & HEAP-SPRAYING

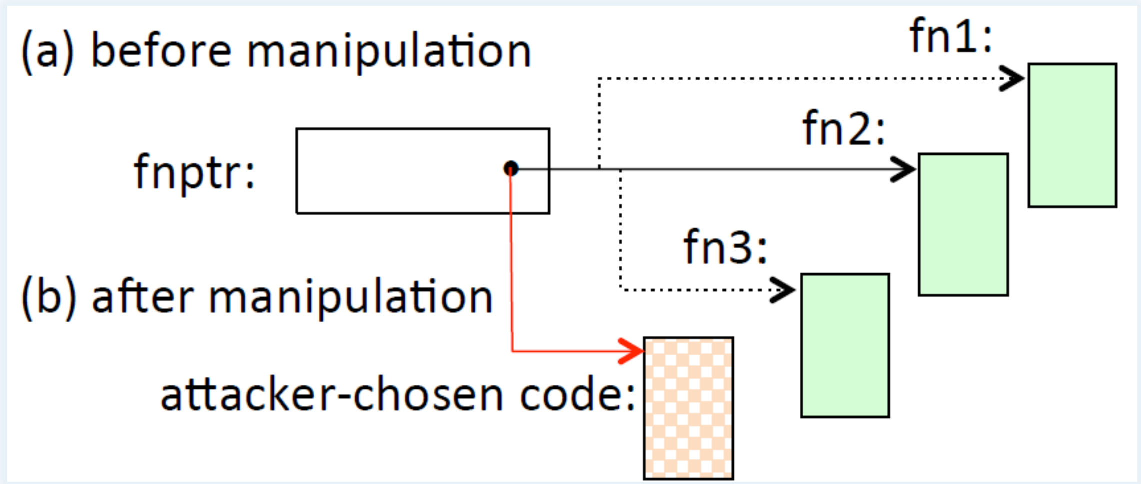
HEAP-BASED BUFFER OVERFLOW



[CREDIT: Oorschot]

OVERFLOWING HIGHER-ADDRESS VARIABLES

- Heap allocations are less predictable than stack ones
  - *attacker must spend time to*
    - find a vulnerable buffer
    - and a useful target to overwrite (without a crash)
  - *e.g., function pointer:*



[CREDIT: Oorschot]

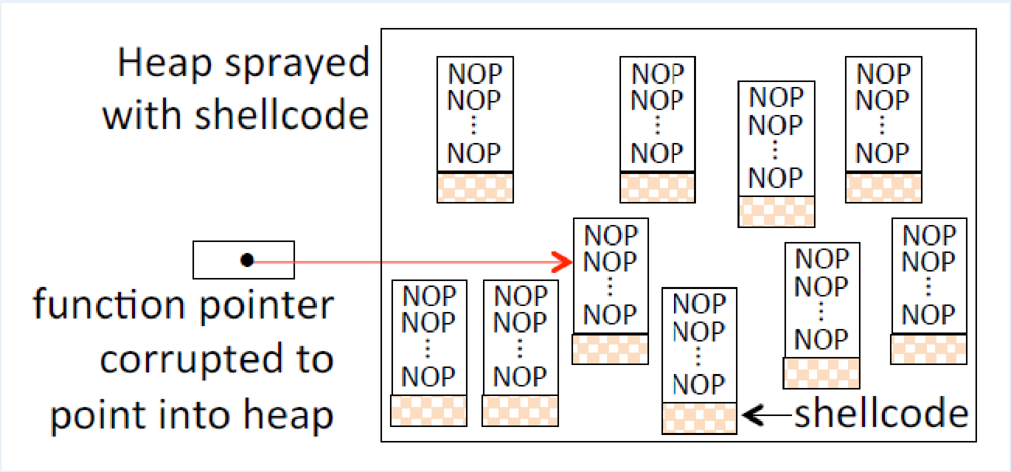
TYPICAL CONTROL FLOW TARGETS

- Stack-based pointers
  - *including return addresses and frame pointers*
- Function pointers (stack/heap/static)
  - *especially including in any function address lookup table*
    - jump table, dispatch table, etc.
- setjmp/longjmp functions addresses
  - *used in exception handling*
- Corrupting data used in a branching test

Exploit steps

- Code injection / code location
  - *chosen code to execute—either injected, or selected from existing code*
- Corruption of control flow data
  - *one or more data structures are overwritten, e.g., by a buffer overflow*
- Seizure of control
  - *program control transferred to exploit code*
    - either needs to be engineered, or wait for it to trigger

HEAP SPRAYING



[CREDIT: Oorschot]

- Place a large number of instances of attacker-chosen code in the heap
  - *could be many thousands*
  - *e.g., used in some browser attacks (triggered by visiting a page)*
    - JavaScript code causes the allocation of 10,000+ strings with chosen content
- Benefit to the attacker
  - *required less precision when executing the corruption of control flow*
    - numerous valid targets

RETURN-TO-LIBC [1]

- General idea
  - » *instead of injecting code, use already existing code part of common libraries*
- Stack-based attack
  - » *stack overflow with* system() *target and chosen parameters*

SYSTEM(3) Linux Programmer's Manual SYSTEM(3)

NAME top

system - execute a shell command

SYNOPSIS top

```
#include <stdlib.h>

int system(const char *command);
```

DESCRIPTION top

The **system**() library function uses fork(2) to create a child process that executes the shell command specified in *command* using execl(3) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

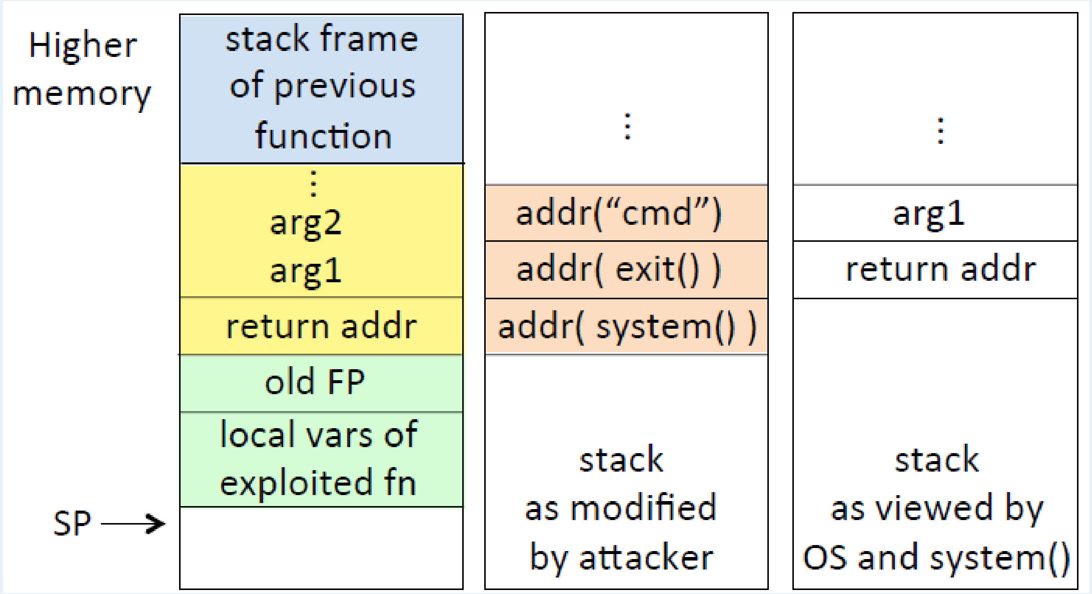
system() returns after the command has been completed.

During execution of the command, **SIGCHLD** will be blocked, and **SIGINT** and **SIGQUIT** will be ignored, in the process that calls **system**(). (These signals will be handled according to their defaults inside the child process that executes *command*.)

If *command* is NULL, then **system**() returns a status indicating whether a shell is available on the system.

RETURN-TO-LIBC [2]

- strcpy of shellcode
  - *attack overwrites stack buffer; now arranged as stack arguments for* strcpy()
  - *injected data includes shellcode to be copied (to the heap)*
  - *stack return address that will be used by* strcpy() *points to shellcode*



[CREDIT: Oorschot]

BUFFER OVERFLOW COUNTERMEASURES [1]

- Compile-time vs. run-time
- Non-executable stack & heap
  - *data execution prevention (NX bit)*
  - *adoption issues and limitations*
    - JIT run-time systems (code *must* execute on the heap)
    - backward compatibility
- Stack protection (run-time)
  - *stack/heap canary*
  - *shadow stacks, pointer protection*

BUFFER OVERFLOW COUNTERMEASURES [2]

- Run-time bounds-checking
  - *compiler support + run-time overhead*
- Address space layout randomization (ASLR, run-time)
  - *randomize layout of objects*
    - stacks, heaps, system libraries
- Type-safe languages
  - *compiler support + run-time checks*
- Safe C libraries
  - *replace* libc*, kill functions like* strcpy
- Static analysis tools (compile-time, binary)
  - *source code analysis*
  - *binary integrity analysis*

BARRIERS TO ADOPTION OF COUNTERMEASURE

- No unified governing body
  - *multiple efforts, standards bodies, “building codes”*
- Backwards compatibility
  - *a great idea with some serious side effects*
- Incomplete solutions
  - *often target only part of the problem*

PRIVILEGE ESCALATION

Forms of escalation

- Examples
  - *moving from the fixed functionality of a compiled program to a shell allowing execution of arbitrary commands and other programs;*
  - *moving from an isolated "sandbox" to having access to a complete filesystem*
  - *moving from a non-root process to code running with UID 0*
  - *moving from UID 0 privileges (user-space process) to kernel-mode privileges*
- Privileged TCP/IP ports
  - *a really bad idea*
  - *https://ar.al/2022/08/30/dear-linux-privileged-ports-must-die/*
    - https://utcc.utoronto.ca/~cks/space/blog/unix/BSDRcmdsAndPrivPorts
- Failures to limit privileged execution
  - *user errors, programmatic errors*