

# **CSC 405**

## **Computer Security**

### **Control Hijacking Attacks**

Alexandros Kapravelos  
[akprav@ncsu.edu](mailto:akprav@ncsu.edu)

# Attacker's mindset

- Take control of the victim's machine
  - Hijack the execution flow of a running program
  - Execute arbitrary code
- Requirements
  - Inject attack code or attack parameters
  - Abuse vulnerability and modify memory such that control flow is redirected
- Change of control flow
  - alter a code pointer (i.e., value that influences program counter)
  - change memory region that should not be accessed

# Buffer Overflows

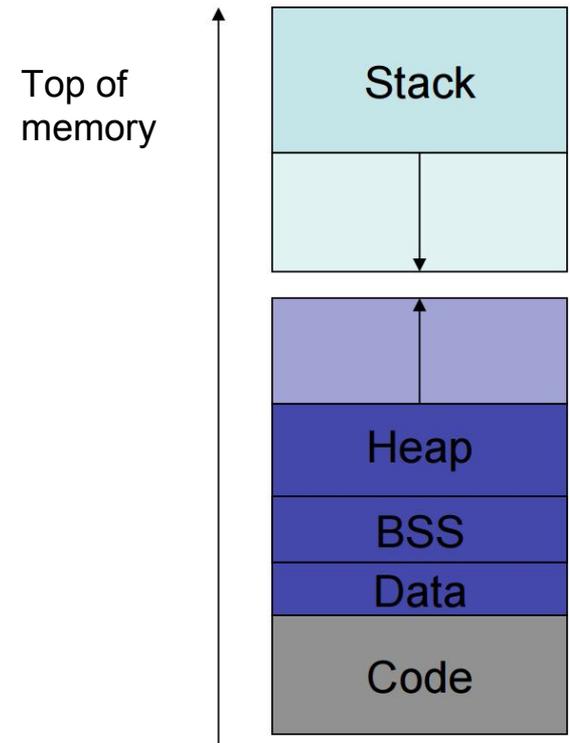
- Result from mistakes done while writing code
  - coding flaws because of
    - unfamiliarity with language
    - ignorance about security issues
    - unwillingness to take extra effort
- Often related to particular programming language
- Buffer overflows
  - mostly relevant for C / C++ programs
  - not in languages with automatic memory management
    - dynamic bounds checks (e.g., Java)
    - automatic resizing of buffers (e.g., Perl)

# Buffer Overflows

- One of the most used attack techniques
- Advantages
  - very effective
    - attack code runs with privileges of exploited process
  - can be exploited locally and remotely
    - interesting for network services
- Disadvantages
  - architecture dependent
    - directly inject assembler code
  - operating system dependent
    - use of system calls
  - some guesswork involved (correct addresses)

# Process memory regions

- Stack segment
  - local variables
  - procedure calls
- Data segment
  - global initialized variables (data)
  - global uninitialized variables (bss)
  - dynamic variables (heap)
- Code (Text) segment
  - program instructions
  - usually read-only



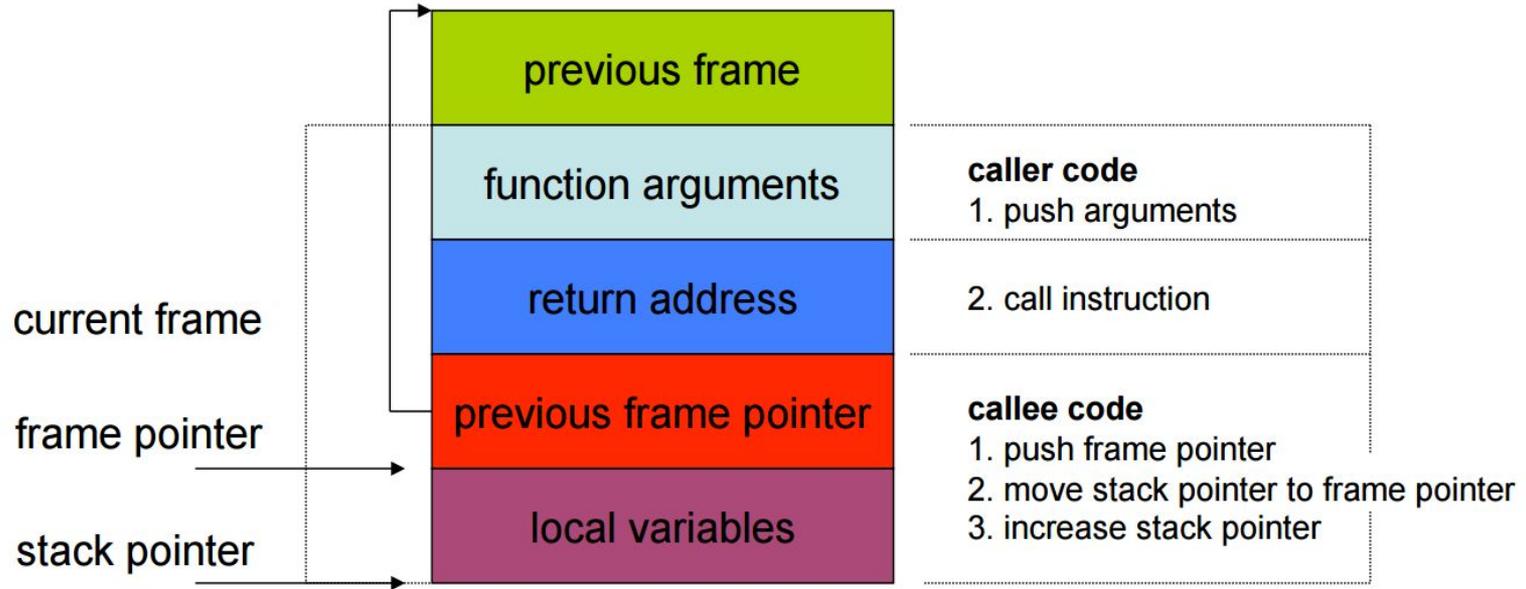
# Overflow types

- Overflow memory region on the stack
  - overflow function return address
  - overflow function frame (base) pointer
  - overflow longjmp buffer
- Overflow (dynamically allocated) memory region on the heap
- Overflow function pointers
  - stack, heap, BSS

# Stack

- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- Processor register points to top of stack
  - stack pointer – SP
  - points to last stack element or first free slot
- Composed of frames
  - pushed on top of stack as consequence of function calls
  - address of current frame stored in processor register
    - frame/base pointer – FP
  - used to conveniently reference local variables

# Stack



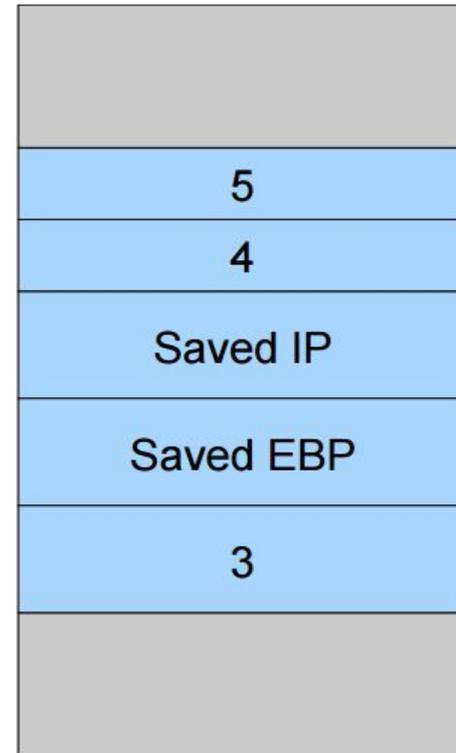
# Procedure Call

```
int foo(int a, int b)
{
    int i = 3;

    return (a + b) * i;
}
```

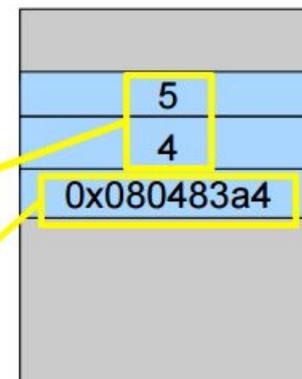


```
int main()
{
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```



# A Closer Look

```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:    push   %ebp
0x0804836e <main+1>:    mov    %esp,%ebp
0x08048370 <main+3>:    sub    $0x18,%esp
0x08048373 <main+6>:    and    $0xffffffff0,%esp
0x08048376 <main+9>:    mov    $0x0,%eax
0x0804837b <main+14>:   add    $0xf,%eax
0x0804837e <main+17>:   add    $0xf,%eax
0x08048381 <main+20>:   shr    $0x4,%eax
0x08048384 <main+23>:   shl    $0x4,%eax
0x08048387 <main+26>:   sub    %eax,%esp
0x08048389 <main+28>:   movl   $0x0,0xffffffff(%ebp)
0x08048390 <main+35>:   movl   $0x5,0x4(%esp)
0x08048398 <main+43>:   movl   $0x4,(%esp)
0x0804839f <main+50>:   call  0x8048354 <foo>
0x080483a4 <main+55>:   mov    %eax,0xffffffff(%ebp)
```



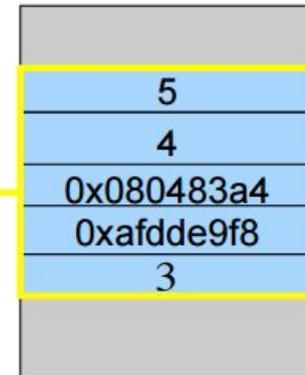
# A Closer Look

```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:   push   %ebp
0x08048355 <foo+1>:   mov    %esp,%ebp
0x08048357 <foo+3>:   sub    $0x10,%esp
0x0804835a <foo+6>:   movl   $0x3,0xffffffff(%ebp)
0x08048361 <foo+13>:  mov    0xc(%ebp),%eax
0x08048364 <foo+16>:  add    0x8(%ebp),%eax
0x08048367 <foo+19>:  imul  0xffffffff(%ebp),%eax
0x0804836b <foo+23>:  leave
0x0804836c <foo+24>:  ret
End of assembler dump.
(gdb)
```

5
4
0x080483a4
0xafdde9f8
3

# The foo Frame

```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8: 0xaf9d3cd8 0x080482de 0xa7faf360 0x00000003
0xaf9d3cd8: 0xafdde9f8 0x080483a4 0x00000004 0x00000005
0xaf9d3ce8: 0xaf9d3d08 0x080483df 0xa7fadff4 0x08048430
```



# Taking Control of the Program

# Buffer Overflow

- Code (or parameters) get injected because
  - program accepts more input than there is space allocated
- In particular, an array (or buffer) has not enough space
  - especially easy with C strings (character arrays)
  - plenty of vulnerable library functions  
`strcpy`, `strcat`, `gets`, `fgets`, `sprintf` ..
- Input spills to adjacent regions and modifies
  - code pointer or application data
    - all the possibilities that we have enumerated before
  - normally, this just crashes the program (e.g., `sigsegv`)

# Example

```
// Test2.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param) {
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char* argv[]) {
    vulnerable(argv[1]);
    printf("Everything's fine\n");
}
```

Buffer that can contain 100 bytes

Copy an arbitrary number of characters from param to buffer

# Let's Crash

```
> ./test2 hello Everything's fine
```

```
> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Segmentation fault
```

```
>
```

# What Happened?

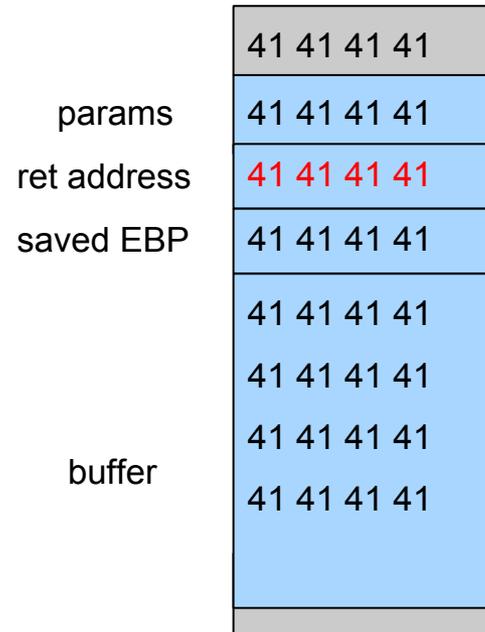
```

> gdb ./test2 (gdb) run hello

Starting program: ./test2 Everything's fine

(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAA...
Program received signal SIGSEGV,
Segmentation fault.
0x41414141 in ?? ()
    
```



# Choosing Where to Jump

- Address inside a buffer of which the attacker controls the content
  - works for remote attacks
  - the attacker need to know the address of the buffer, the memory page containing the buffer must be executable
- Address of a environment variable
  - easy to implement, works with tiny buffers
  - only for local exploits, some programs clean the environment, the stack must be executable
- Address of a function inside the program
  - works for remote attacks, does not require an executable stack
  - need to find the right code, one or more fake frames must be put on the stack

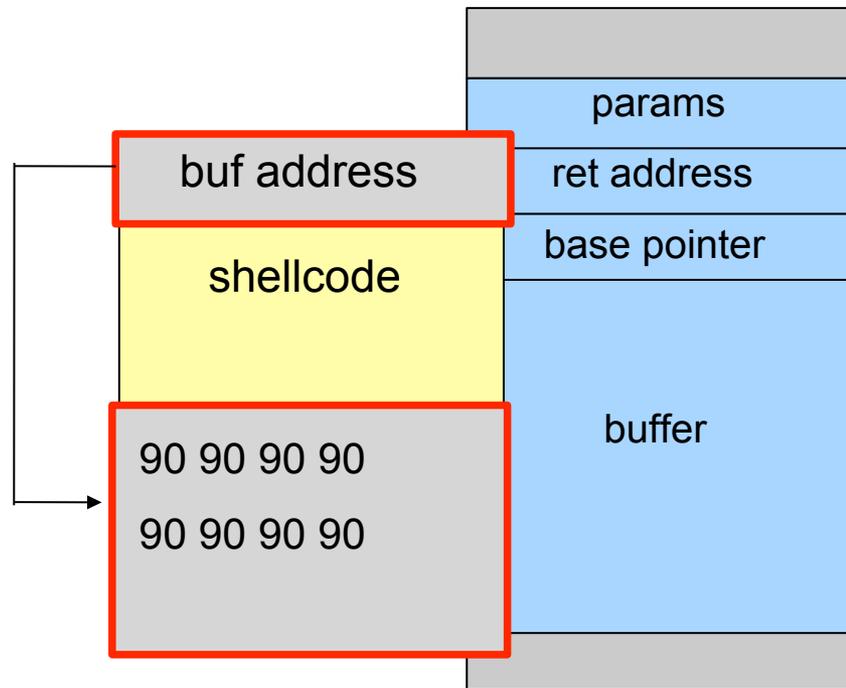
# Jumping into the Buffer

- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
  - The address must be precise: jumping one byte before or after would just make the application crash
  - On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine
  - Any change to the environment variables affect the stack position

# Solution: The NOP Sled

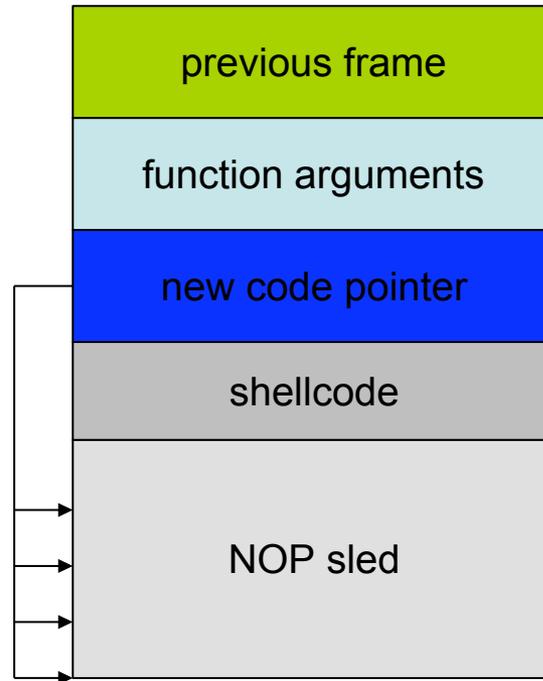
- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever the program jump into it..
  - .. it always finds a valid instruction
  - .. it always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of no operation (NOP) instructions
  - single byte instruction (0x90) that does not do anything
  - more complex sleds possible (ADMmutate)
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

# Assembling the Malicious Buffer



# Code Pointer

Any return address into  
the NOP sled succeeds



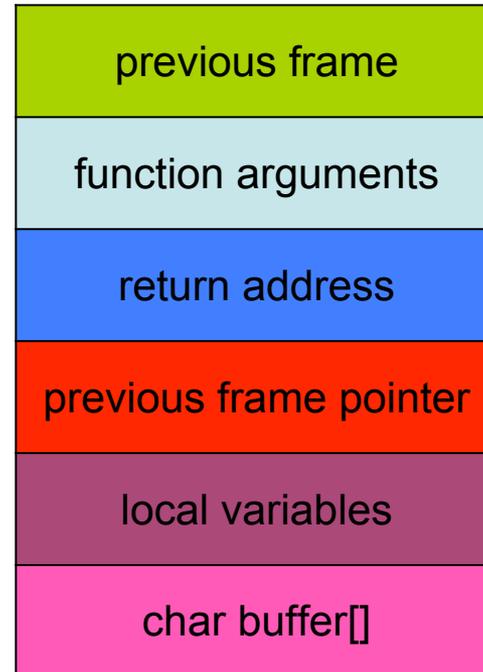
# Solution: Jump using a Register

- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)
- Locate an instruction that jump/call using that register
  - can also be in one of the libraries
  - does not even need to be a real instruction, just look for the right sequence of bytes
  - you can search for a pattern with gdb find

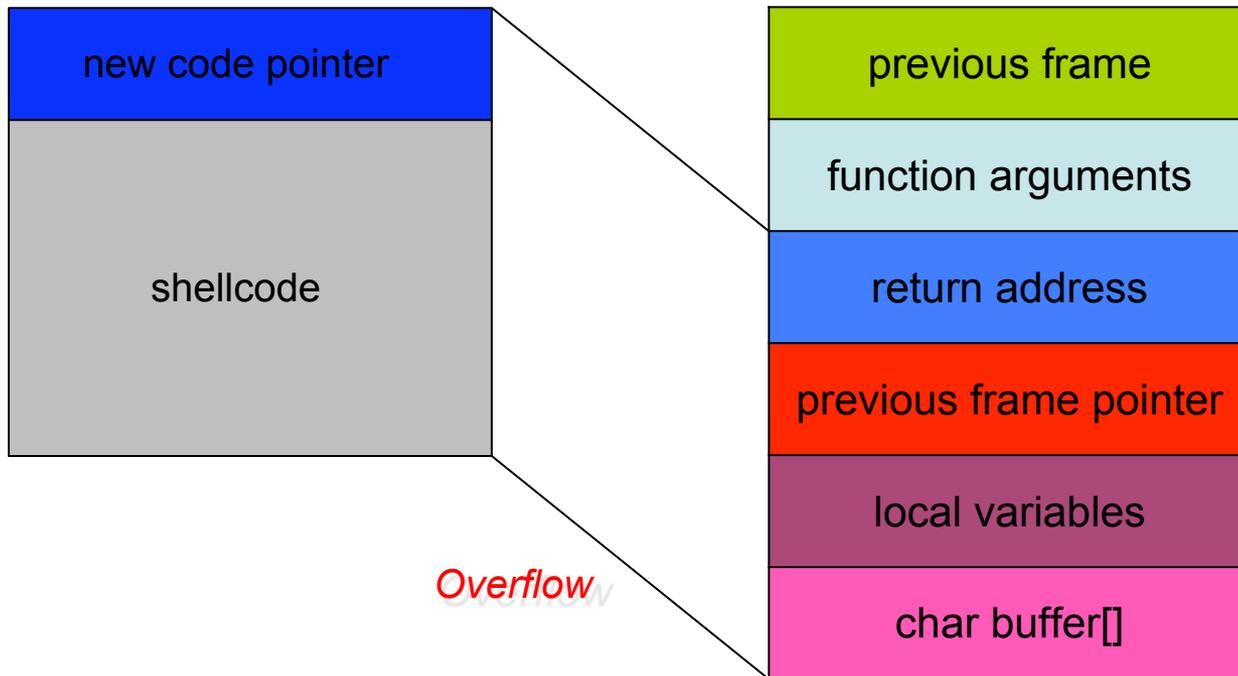
```
jmp ESP = 0xFF 0xE4
```

- Overwrite the return address with the address of that instruction

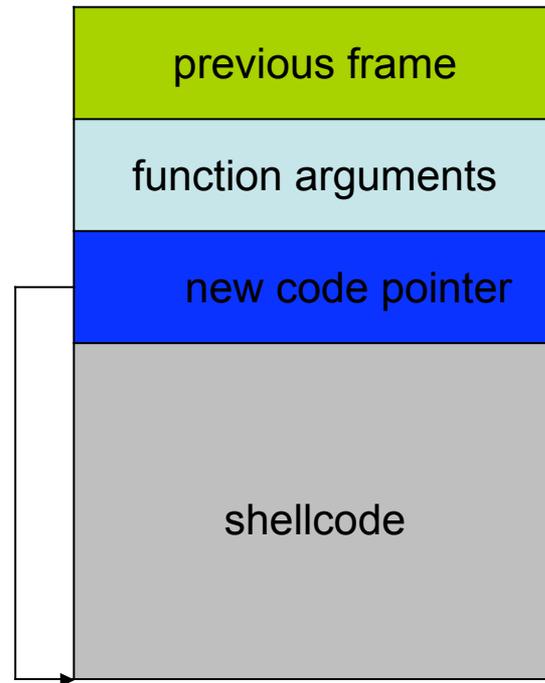
# Pulling It All Together



# Pulling It All Together



# Pulling It All Together



# Small Buffers

- Buffer can be too small to hold exploit code
- Store exploit code in environmental variable
  - environment stored on stack
  - return address has to be redirected to environment variable
- Advantage
  - exploit code can be arbitrary long
- Disadvantage
  - access to environment needed

# Format String Vulnerability

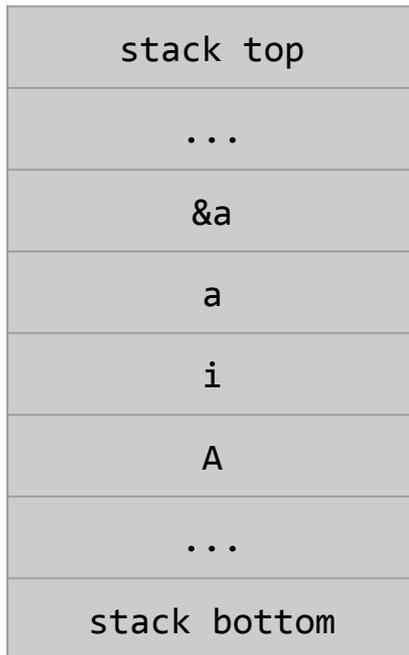
- Problem of user supplied input that is used with `*printf()`
  - `printf("Hello world\n");` // is ok
  - `printf(user_input);` // vulnerable
- `*printf()`
  - function with variable number of arguments
  - `int printf(const char *format, ...)`
  - as usual, arguments are fetched from the stack
- `const char *format` is called format string
  - used to specify type of arguments
    - `%d` or `%x` for numbers
    - `%s` for strings

# Format string

parameter	output	passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

# The stack and its role at format strings

```
printf("Number %d has no address, number %d has: %08x\n", i, a, &a);
```



A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

# Format String Vulnerability

```
#include <stdio.h>

int main(int argc, char **argv) {
    char buf[128];
    int x = 1;

    snprintf(buf, sizeof(buf), argv[1]);
    buf[sizeof(buf) - 1] = '\0';

    printf("buffer (%d): %s\n", strlen(buf), buf);
    printf("x is %d/%#x (@ %p)\n", x, x, &x);
    return 0;
}
```

# Format String Vulnerability

```
$ ./vul "AAAA %x %x %x %x"  
buffer (28): AAAA 40017000 1 bffffff680 4000a32c  
x is 1/0x1 (@ 0xbffffff638)
```

```
$ ./vul "AAAA %x %x %x %x %x"  
buffer (35): AAAA 40017000 1 bffffff680 4000a32c 1  
x is 1/0x1 (@ 0xbffffff638)
```

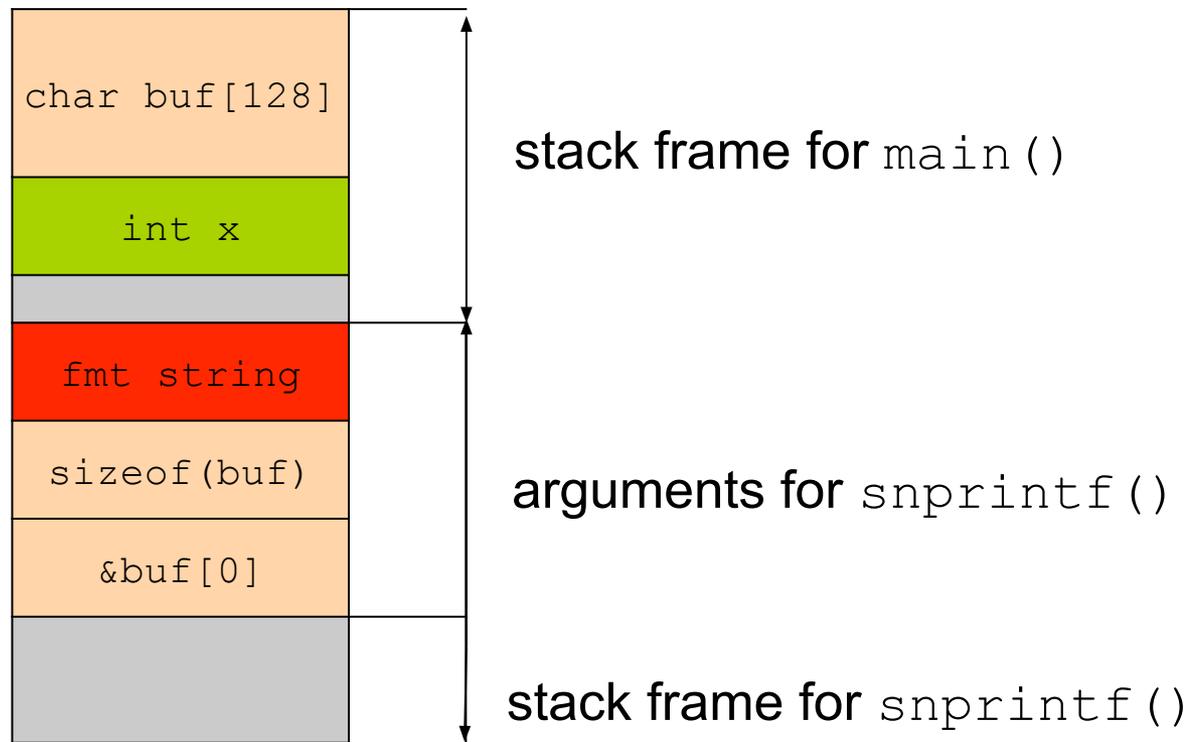
```
$ ./vul "AAAA %x %x %x %x %x %x"  
buffer (44): AAAA 40017000 1 bffffff680 4000a32c 1 41414141  
x is 1/0x1 (@ 0xbffffff638)
```

**We are pointing to our format string itself!**

What happens when a format string does not have a corresponding variable on the stack?

# Format String Vulnerability

Stack Layout



# Format String Vulnerability

```
$ ./vul $(python -c 'print "\x38\xf6\xff\xbf %x %x %x %x %x %x"')
buffer (44): 8öÿ; 40017000 1 bffff680 4000a32c 1bffff638
x is 1/0x1 (@ 0xbffff638)
```

```
$ ./vul $(python -c 'print "\x38\xf6\xff\xbf %x %x %x %x %xn"')
buffer (35): 8öÿ; 40017000 1 bffff680 4000a32c 1
x is 35/0x2f (@ 0xbffff638)
```

# Format String Vulnerability

- **%n**

The number of characters written so far is stored into the integer indicated by the int\*(or variant) pointer argument

- One can use width modifier to write arbitrary values

- for example, `%.500d`
- even in case of truncation, the values that would have been written are used for `%n`

- More resources

- <https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf>
- <https://www.exploit-db.com/docs/28476.pdf>

# Your Security Zen

## MikroTik RouterOS exploit in the wild

TCP SCANS ON PORT 8291

