

## Midterm Study Guide

Sunday, October 22, 2017 6:58 PM

### CS363 Information Assurance and Security

#### -- Midterm Exam Study Guide

*(I reserve the right to interpret if a question is covered by this guide. Still, only about 90% of questions are covered by this guide.)*

#### Overview:

- ✓ 1. CIA.
- ✓ • Can rephrase it.
- ✓ • Can give application examples.

#### Buffer overflow:

- ✓ 1. Can explain what it is, the cause of it, how to implement it
- ✓ 2. Know stack-based, heap-based, and in other segments
- ✓ 3. Can identify buffer overflow in the source code

#### Shellcoding:

- ✓ 1. Can explain how to use shellcode to exploit buffer overflow attacks
- ✓ 2. Know the process to write shellcode

#### Format string:

- ✓ 1. Know how to use format string to exploit buffer overflow attacks
- ✓ 2. Can identify format string security issues in the source code

#### Countermeasures:

- ✓ 1. Know defenses and countermeasures: canary, shadow stack, non-executable bit, ISR, ASR
- ✓ 2. Can use defenses and countermeasures to analyze scenarios.

### Overview:

#### 1) CIA

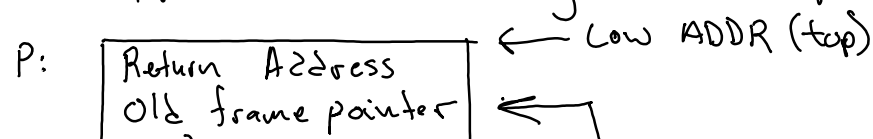
Confidentiality  
Integrity  
Availability

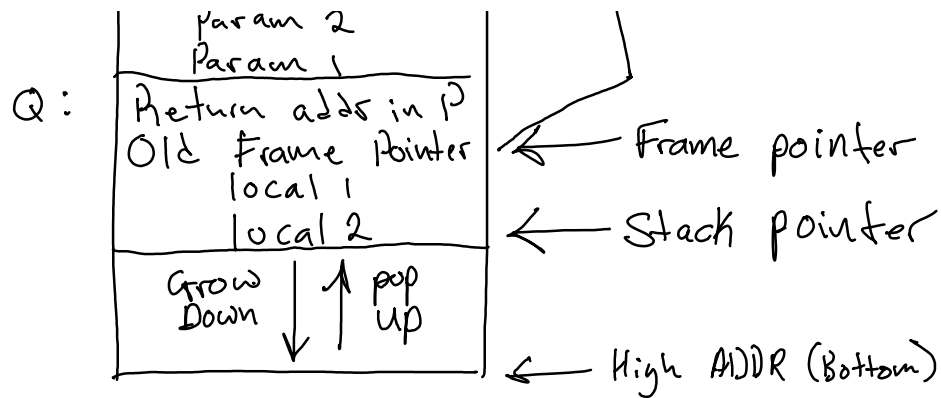
### Buffer Overflow:

- 1) Definition: an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

- 2) Stack Buffer Overflows: occurs when the targeted buffer is located on the stack, usually as a local variable in a function's stack frame.  
AKA Stack smashing

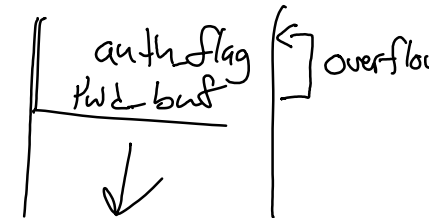
Stack frame: P calling Q (Pop-up, grow Down)





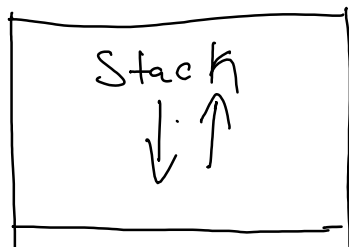
```

int check_authentication (char * password) {
  int auth_flag = 0;
  char password_buffer[16]; // 17+ chars = buf overflow
  strcpy(password_buffer, password);
  if (strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
  if (strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
  return auth_flag;
}
  
```



Stack Buffer Overflow used to crash system or execute code / Inject ADDR

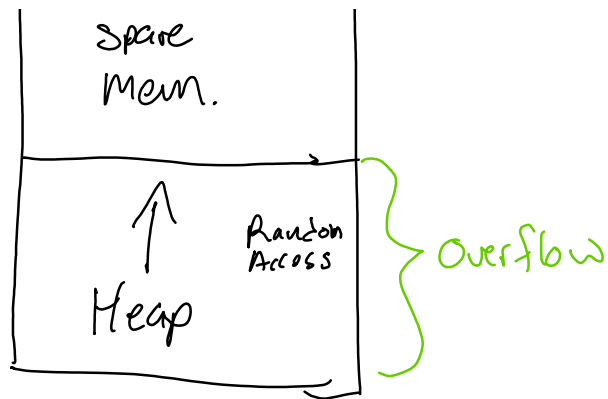
Heap Overflow: Overflow on heap ... Doh.



Pg 369 for struct "chunk"

```

int main (int argc, char * argv[])
{
  chunk_t *next;
  setbuf(stdin, NULL);
}
  
```

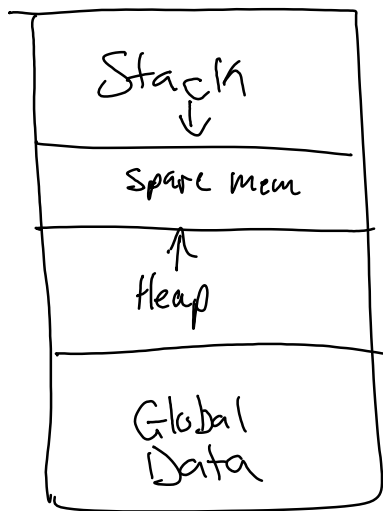


```

next = malloc(sizeof(chunk_t));
next->process = showLen;
printf("Enter value: ");
gets(next->inp);
next->process(next->inp);
printf("buffers done\n");
}

```

Global Data overflow: buffers located in the program's static/global data area



```

/* Global Static Data -> Struct chunk */
int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showLen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffers done\n");
}

```

3) Common Buffer Overflow weak points in Source Code:

`gets(char *str)` → read line from standard input into str

`sprintf(char *str, char *format, ...)` → create str according to supplied format and vars

`strcat (char *dest, char *src)` → append contents of string src to string dest

`strcpy (char *dest, char *src)` → copy contents of string src to src dest

`vsprintf (char *str, char *fmt, va_list ap)` → create str according to supplied format and var:

Shell coding:

1) Shell code → used to transfer execution to code supplied by the attacker → Assembly

NOP sled → code placed near end of buffer and padded with NOP instructions



Buffer

NOP Sled:

```
memset(buffer, 0x90, 60); // Build NOP Sled (60 bytes)
```

Shell code:

```
char shellcode[] =
  "\x31\x03\x03\xdb\x81\xc9\x99...";
```

↓

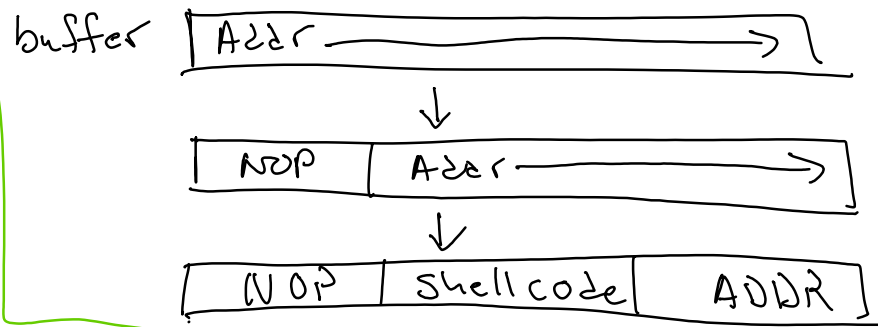
```
memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
```

Repeated ADDR:

```
int i;
for (i = 0; i < 1000; i++)
  printf("%s", "A");
```

ret - (unsigned int) 0 - offset, // set return Addr

for (i=0; i < 160, i+=4) // fill buffer w/ return Addr  
 \*((unsigned int \*) (buffer + i)) = ret;



2) Shellcode = a string of assembly instructions for the OS

Process of writing SC:

- 1) determine Software attcking
- 2) determine OS Software resides on
- 3) determine CPU architecture (what assembly must I use)
- 4) compose desired command in corresponding assembly

↓  
 90 90 eb 1a ← x86 machine code

↓  
 "\x90\x90\xeb\x1a" ← x86 shellcode

format String overflow:

printf(text); ← NO! BAD! Vulnerable!

printf("%s", text); ← YES! CORRECT WAY!

By passing a string with formatting tags the bad way will try to evaluate the tag.

perl -e 'print "%08x." x 40' ← this will print ADDRs (stack)  
if bad printf uses  
don't forget, little endian (backwards ADDR)

bytes 0x25, 0x30, 0x38, 0x78, and 0x2e repeats

Printf "\x25\x30\x38\x78\x2e\n" → %08x.  
Stack ADDR?!

%s can read strings from mem addresses

%n can overwrite mem address

perl -e 'print "AAAA" + "\x?\x?" + "BBBB" + "%x\n" x 127 + "%n" ~ 'x08'

target ADDR  
padding to ID  
ADDR offset  
write to ADDR @ offset  
if %s leave of the  
-1 x 08!  
and read @  
wri

exploit - exercises.com

Some practice

Counter measures:

Canary:

foo(i)  
{  
... not in heap or stack

```

static int canary = rand();
int mySecret;
mySecret = canary

```

```

...
if (mySecret != canary) { alert(); } ← if canary finds "poison" error out
exit(0);
}

```

Cannot protect against format string attack (Arb. ADDR  $\Rightarrow$  Arb. value)

Shadow Stack:

Separate stack to hold <sup>copies</sup> return addresses for comparison  
no protection for other data

W $\oplus$ X / DEP:

Make all writable memory as non-Executable  
 $\hookrightarrow$  MS DEP (Data Execution Prevention)  
 Blocks all code inject exploits

Hardware Support: AMD "NX" bit, Intel "XD" bit (post 2004 CPU)

Widely deployed: Windows (since xp sp2), Linux (via PaX patches),  
 OpenBSD, OS X (since 10.5)

Problems:

we can do (return-to-libc) → 1.0 functions where we do  
 → overwrite to libc with

Overwrite return address w/  
 address of libc function

- Setup fake return address and arguments
- ref will "call" libc function

## No Injected Code

### Defense Summary

- No Defense method against all attacks

ISR → Instruction Set Randomization

Randomized IS locations

can't protect against libc attack unless the libc  
 funct addrs are also randomized (ASLR → Address Spaces Randomization)

### ASLR

Randomize memory addresses

Attacks to ASLR

Address Guessing for libc (probe for ADDR)  
 probe for offset  
 libc attack



Better ASK:

- 64-bit arch. (larger mem space)
- ↳ increase randomness
- Randomization Frequency (Rand. inside  $\rightarrow$  vars)

Granularity

- permute stack vars
- permute code & lib functions
- permute static data

combine w/ other approaches. (ISR + ASR)

GDR:

- i) compile so can use gdb on code  
↳ gcc [extra flags] -g <source code> -o <output file>

## 2) GDB commands

Break Point: `break` `<source> : <Line>`  
or  
`break` `<function name>`

Continue/Step

print case: list

```
print assembly, disassemble <func>
```

Run Code : run

show contrast in info reader receives

show content in  
all gen. purpose reg: info register

examine memory: X <addr>

i - instruction

example: x/2i \$eip

### 3) Registers

General Purpose:

Temp variables { EAX → Accumulator  
ECX → counter  
EDX → Data  
EBX → Base

Pointers & Indexes { ESP → Stack Pointer  
EBP → Base Pointer  
ESI → Source Index  
EDI → Destination Index

EIP → Instruction Pointer

EFLAGS: bit flags

↳ for comparisons and  
memory segmentation