CSE 127 Discussion
Week 3

Ariana Mirian
April 14, 2020
Zoom
This discussion is being recorded
Logistics

• PA1 due yesterday...still have late days if needed
• PA2 released yesterday on webpage
  • Due April 23rd (next Thursday) at 12:30 PM Pacific time
• Partners!
  • If you want a partner you need to stick with them
  • Partners are for helping each other out!
  • Please fill out google form by Friday April 17th EOD
    • https://forms.gle/mwQ9EpknejtVHLoe8
    • Link also on Piazza
• Today
  • Buffer Overflow Basics
  • PA2 overview
  • Open Office hours
Buffer Overflow Review

• Format string vulnerabilities
• Heap vulnerabilities
• Integers
Format String Vulnerabilities

• `printf()`
  • What’s the problem with `printf`?
• Variadic function – variance in what can be input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int)</td>
<td>value</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%s</td>
<td>string ((const) (unsigned) char *)</td>
<td>reference</td>
</tr>
<tr>
<td>%n</td>
<td>number of bytes written so far, (* int)</td>
<td>reference</td>
</tr>
</tbody>
</table>
Format String Vulnerabilities

- `printf()`
  - What’s the problem with `printf`?
- Variadic function – variance in what can be input
- What do the follow `printf` vulnerabilities do?
  - `printf("\x10\x01\x48\x08 %x %x %x %x %s")`
  - `printf("%s\n")`
  - `printf("%08x %08x %08x %08x %08x\n")`
Format String Vulnerabilities

• printf()
  • What’s the problem with printf?
• Variadic function – variance in what can be input
• printf("\x10\x01\x48\x08 %x %x %x %x %x %s")
  • will print out the content’s in the memory address 0x10014808
Format String Vulnerabilities

• `printf()`
  • What’s the problem with `printf`?

• Variadic function – variance in what can be input

• `printf("\x10\x01\x48\x08 %x %x %x %x %s")`
  • will print out the content’s in the memory address 0x10014808

• `printf("%s\n")`
  • Take the previous stack word, interpret as pointer (reference), and print memory at address as string
Format String Vulnerabilities

• `printf()`
  • What’s the problem with `printf`?

• Variadic function – variance in what can be input

• `printf("%x %x %x %x %s")`
  • will print out the content’s in the memory address 0x10014808

• `printf(“%s
”)`
  • Take the previous stack word, interpret as pointer (reference), and print memory at address as string

• `printf("%08x %08x %08x %08x
”)`
  • Retrieve five parameters from the stack, display as 8digit padded hexadecimal numbers
Heap Vulnerabilities

• Dynamically allocated memory in program
• Programmer is responsible for many of the details
  • Variable liveliness and validity
  • In stack this is not the case
• Heap are kept in doubly-linked lists (bins)
• malloc() and free() are common commands that can get a user in trouble
  • Trouble because of differences in expectation vs reality
• Double free and use after free
Heap Vulnerabilities

• Dynamically allocated memory in program
• Programmer is responsible for many of the details
  • Variable liveliness and validity
  • In stack this is not the case
• Heap are kept in doubly-linked lists (bins)
• malloc() and free() are common commands that can get a user in trouble
  • Trouble because of differences in expectation vs reality
• Double free and use after free
Free List

- Free chunks are kept in circular doubly-linked lists (bins)
Heap Vulnerabilities

• Dynamically allocated memory in program
• Programmer is responsible for many of the details
  • Variable liveliness and validity
  • In stack this is not the case
• Heap are kept in doubly-linked lists (bins)
• malloc() and free() are common commands that can get a user in trouble
  • Trouble because of differences in expectation vs reality
• Double free and use after free
Double Free

```c
a = malloc(10);    // 0xa04010
b = malloc(10);    // 0xa04030
c = malloc(10);    // 0xa04050

free(a);
free(b);          // To bypass "double free or corruption (fasttop)" check
free(a);          // Double Free !!

d = malloc(10);    // 0xa04010
e = malloc(10);    // 0xa04030
f = malloc(10);    // 0xa04010  - Same as 'd'!
```

https://heap-exploitation.dhavalkapil.com/attacks/double_free.html
Double Free

```c
a = malloc(10); // 0xa04010
b = malloc(10); // 0xa04030
c = malloc(10); // 0xa04050

free(a);
free(b); // To bypass "double free or corruption (fasttop)" check
free(a); // Double Free !!

d = malloc(10); // 0xa04010
e = malloc(10); // 0xa04030
f = malloc(10); // 0xa04010 - Same as 'd' !
```

https://heap-exploitation.dhavalkapil.com/attacks/double_free.html
a = malloc(10); // 0xa04010
b = malloc(10); // 0xa04030
c = malloc(10); // 0xa04050

free(a);
free(b); // To bypass "double free or corruption (fasttop)"
free(a); // Double Free !!

d = malloc(10); // 0xa04010
e = malloc(10); // 0xa04030
f = malloc(10); // 0xa04010 - Same as 'd'!

The state of the particular fastbin progresses as:

1. 'a' freed.
   head -> a -> tail

2. 'b' freed.
   head -> b -> a -> tail

3. 'a' freed again.
   head -> a -> b -> a -> tail

https://heap-exploitation.dhavalkapil.com/attacks/double_free.html
a = malloc(10); // 0xa04010
b = malloc(10); // 0xa04030
c = malloc(10); // 0xa04050

free(a);
free(b); // To bypass "double free or corruption (fasttop)"
free(a); // Double Free !!

d = malloc(10); // 0xa04010
e = malloc(10); // 0xa04030
f = malloc(10); // 0xa04010 - Same as 'd'!

The state of the particular fastbin progresses as:

1. 'a' freed.
   - head -> a -> tail

2. 'b' freed.
   - head -> b -> a -> tail

3. 'a' freed again.
   - head -> a -> b -> a -> tail

4. 'malloc' request for 'd'.
   - head -> b -> a -> tail ['a' is returned]

5. 'malloc' request for 'e'.
   - head -> a -> tail ['b' is returned]

6. 'malloc' request for 'f'.
   - head -> tail ['a' is returned]
Integer Overflow/Conversion

• Bounds check (types)
  • What if we take a signed long and truncate it into an unsigned short?
  • This changes the value
  • 0010 0000 0000 0000 0000 0000 0000 0000 →
  • 0000 0000 0000 0000

0010 0000 0000 0000 0000 0000 0000 0000 →
0000 0000 0000 0000
Integer Overflow/Conversion

- **Signed vs unsigned**
  - Signed: MSb is the sign bit (0 is positive, 1 is negative)
  - Unsigned: MSb is just the largest bit

- **Example 0xFFFFFFFF (0xF = 1111)**
  - Signed: -1
  - Unsigned: 4294967295
Integer Overflow/Conversion

• Exercise: take two minutes and write out the unsigned and signed versions of a 4 bit integer from 0000 to 1111
<table>
<thead>
<tr>
<th>Unsigned bits value</th>
<th>Signed bits value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0</td>
<td>0000 0</td>
</tr>
<tr>
<td>0001 1</td>
<td>0001 1</td>
</tr>
<tr>
<td>0010 2</td>
<td>0010 2</td>
</tr>
<tr>
<td>0011 3</td>
<td>0011 3</td>
</tr>
<tr>
<td>0100 4</td>
<td>0100 4</td>
</tr>
<tr>
<td>0101 5</td>
<td>0101 5</td>
</tr>
<tr>
<td>0110 6</td>
<td>0110 6</td>
</tr>
<tr>
<td>0111 7</td>
<td>0111 7</td>
</tr>
<tr>
<td>1000 8</td>
<td>1000 -8</td>
</tr>
<tr>
<td>1001 9</td>
<td>1001 -7</td>
</tr>
<tr>
<td>1010 10</td>
<td>1010 -6</td>
</tr>
<tr>
<td>1011 11</td>
<td>1011 -5</td>
</tr>
<tr>
<td>1100 12</td>
<td>1100 -4</td>
</tr>
<tr>
<td>1101 13</td>
<td>1101 -3</td>
</tr>
<tr>
<td>1110 14</td>
<td>1110 -2</td>
</tr>
<tr>
<td>1111 15...</td>
<td>1111 -1</td>
</tr>
</tbody>
</table>
Integer Overflow/Conversion

• Helps us do some fun things with multiplication too...

\[ 4160749577 \times 32 = 4160749557 \times 2^5 = \]
\[ 4160749577 \ll 5 \]

\[ 4160749577 = \]
\[ 0b1111100\ldots1001 \]
\[ \ll 5 = 0b00\ldots100100000 \]
\[ = 1(32) + 1(256) = 288 \]
Integer Overflow/Conversion

• Helps us do some fun things with multiplication too...

\[
4160749577 \times 32 = 4160749577 \times 2^5 = 4160749577 \ll 5
\]

Notice the MSb!
What unsigned number can we multiply by 32 to get (the same answer)?

\[
4160749577 = 0b1111100\ldots1001
\ll 5 = 0b00\ldots100100000 = 1(32) + 1(256) = 288
\]
Integer Overflow/Conversion

• Helps us do some fun things with multiplication too...

4160749577 * 32 = 4160749557 * 2^5 = 4160749577 << 5

4160749577 = 0b1111100.....1001
<< 5 = 0b00.....100100000
= 1(32) + 1(256) = 288

Notice the MSb!
Unsigned: 4160749577
Signed: -134217719

-134217719 * 32 = 288
PA2 Overview

• Going to produce buffer overflow exploits
  • Going to put theory to practice!
  • Won’t be working on countermeasures
  • Four exploits in total
    • #1 – 3 are stack based
    • #1 is based off of Aleph One’s paper
    • #4 is heap based (and extra credit!)

• `generatesrc.py` generates target using the base and is randomized using YOUR PID
  • Follow instructions on the assignment in order!

• Your buffer sizes and offsets will differ from everyone else’s
Setting

- target[1-4].c are vulnerable pieces of code that each read a string from the command line
- Your exploit is the string you pass in
- COULD run the attack by running $ ./target1 <attack_string>
- But that’s difficult
  - Hard to type the string and fix things in place
  - Some of the strings could be very long
- So we call the strings from C programs called sploit[1-4].c
- sploit[1-4].c is the programmatic version of calling ./target[1-4] from the command line
- ONLY MODIFY SPLOIT[1-4].c. DO NOT CHANGE TARGET.
Shellcode

```c
#include <stdio.h>

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

http://phrack.org/issues/49/14.html#article

```c
static char shellcode[] =
"\xeb\xf1f\xe5e\x89f\xb6\x9f\x08\x31c0\x88\x46\x07\x89f\x4e\x0c\x0b0b0b\n"\x89f\xe3f\x8d\xe4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\xb9f\x840\xcd\n"\x80\xe8\xcdc\xff\xff\xff/bin/sh";
```
Setuid

• Bit that allows elevation of privilege
• The targets run as their owner (root)
• Student can execute as root as long as it’s executing ‘target[1-4]’
• Root shell!
  • Student can now run ‘rm –rf /’ (but maybe don’t do this 😃 )
General Workflow

• 1) Find the vulnerability (one of the examples from lecture or discussion)

• 2) Create an exploit to use that vulnerability
  • exploit the difference Expectation vs. reality
include <stdio.h>
#include <stdlib.h>
#include <string.h>

int bar(char *arg, char *out)
{
    strcpy(out, arg);
    return 0;
}

int foo(char *argv[])
{
    char buf[768];
    bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}
What is the exploit?
(you write this!)
Tips and Tricks

• Avoid 0x00
  • You can’t null terminate!

• 0x90 NOP
  • NOP sled
  • Good for padding

• memcpy and loops are your friends
  • Don’t write an 800 byte buffer by hand

• Refer to Aleph One for sploits 1-2
Questions to ask yourself

• How can I get the program to return to an address I control? What do I need to overwrite?

• What if I write X more bytes than the buffer allows? Where does that put me in the stack/heap?

• Where in the program is there a bounds check? Can I get past the bounds check?

• How can I use negative numbers or signed/unsigned ints to mask my end goal?
Open Office Hours & Questions