Tool Developer Qualification Course

Reverse Engineering project

Bomb

Joshua Abraham

Taught by by
Dr. Joseph Santmeyer

November 17, 2017

Contents

1 Summary

I began my analysis by reading the assembly of each stage to get a general idea of what they were doing. This static analysis is my preferred way to solve challenges like these. Some of the assembly code snippets in this document has been edited to be more readable (i.e. identifying local variables with a name like "input" instead of an rbp offset).

```
gdb -batch -ex 'file ./bomb.patched' -ex 'disassemble /r stage_1' > objectdumps/stage_1.out
....
gdb -batch -ex 'file ./bomb.patched' -ex 'disassemble /r stage_8' > objectdumps/stage_8.out
```

While performing dynamic analysis with the GNU Debugger (gdb), the program pretends to segfault by detecting that it is being debugged. The program uses ptrace to do so, a common anti-reverse engineering tactic. While the majority of my analysis of the code was static, just reading the code and translating to higher level constructs, I occasionally needed to perform tests with the binary. To avoid having to manually step over the call to ptrace in _start every single time, I patched the binary with gdb to call main *if it was being debugged*, instead of if it was not. I did so with the following commands:

```
cp bomb bomb.patched
gdb -write -q ./bomb.patched
disassemble /r _start
....
0x0000000000401461 <+49>:    75 60    jne    0x4014c3
....
set {unsigned char}0x0000000000401461 = 0x74
disassemble /r _start
....
0x0000000000401461 <+49>:    74 60    je     0x4014c3
....
quit
```

Using an x64 assembler I found that the je and jne instructions are the same size, and only differ by one byte. We can skip the fake segfault message by making a patched copy of the bomb binary that will run in a debugger.

1. Stage 1: swordfish
2. Stage 2: jabraham
3. Stage 3: 1872
4. Stage 4: 107 214 428 856 1713
5. Stage 5:   1172
6. Stage 6: 48 a 48
7. Stage 7: (press enter)
8. Stage 8: (run ./stag8.sh in the same directory as the binary, then press enter).

2 Stage 1 Analysis

Analysis:

Stage 1 is straightforward. Below is an abbreviated listing of the relevant disassembled code:

```
mov     esi, 0x4018f8   ; "swordfish"
mov     rdi, rax        ; user input
call    strcmp
test    eax, eax
setz    al
leave
```

This stage calls strcmp with user input and the string "swordfish". If the return value is zero (meaning the strings are identical), this stage returns non-zero and the program proceeds.

Pseudocode:

```
bool stage_1(char *user_input)
{
  return strcmp(user_input, "swordfish") == 0;
}
```

Solve Script:

Since there was no math needed to calculate my password for this stage, no script was needed.

3 Stage 2 Analysis

Analysis:

Stage 2 starts with a call to getenv. This function takes one parameter and returns the value of a specified environment variable.

```
mov     edi, 0x401902     ; "USER"
call    getenv
mov     QWORD PTR [rip+0x201264], rax
```

The user name is stored in a global variable that is used throughout the binary at rip+0x201264. In my case the user name is "jabraham". This and the saved user input are passed to strcmp, similarly to Stage 1.

```
mov     rax, [rbp-0x8]
mov     rsi, rdx          ; "jabraham"
mov     rdi, rax          ; "jabraham"
call    strcmp
test    eax, eax
setz    al
leave
```

After the call to strcmp, the program checks the return value in eax and sets al if the zero flag is set. The stage is passed if the user supplies their own username.

Pseudocode:

```
bool stage_2(char *user_input)
{
  username = getenv("USER");
  return strcmp(user_input, username) == 0;
}
```

Solve Script:

As with Stage 1, there was no need to perform calculations to find the password for this stage.

4 Stage 3 Analysis

Analysis:

Stage 3 takes user input and saves it in [rbp-0x28]. The global variable holding the username is moved into [rbp-0x8].

```
push     rbp
mov      rbp, rsp
sub      rsp, 30h
mov      [rbp-0x28], rdi
mov      rax, fs:28h
mov      [rbp-0x08], rax
xor      eax, eax
mov      rcx, cs:src
mov      rax, [rbp-0x28]
```

Once these variables are set, they are passed to the C function strncat. It takes three parameters: the destination, the source, and the maximum number of bytes to concatenate.

```
mov      edx, 0x80          ; number
mov      rsi, rcx           ; source (username)
mov      rdi, rax           ; destination (input)
call     strncat
```

The destination will now look similar to this: "100jabraham". The bomb then calls strlen on the original username variable.

```
mov      rax, [rbp+0x28]
mov      rdi, rax           ; username ("jabraham")
call     strlen
```

The length of the username is saved on the stack at [rbp-0x10]. Then the bomb gets ready for a call to sscanf. This function scans a string using a format string and stores each match in following parameters. Here sscanf is passed the concatenated string, the format specifier "%u", and the address of a stack variable. The "%u" tells us that sscanf is looking for a number, meaning the password for this stage is a single number.

```
mov      [rbp-0x10], rax
mov      [rbp-0x14], 0
lea      rdx, [rbp-0x14]
mov      rax, [rbp-0x28]
mov      esi, 0x401907    ; "%u"
mov      rdi, rax
mov      eax, 0
call     sscanf
```

Next the binary performs series of calculations with the total length and input number. Note: for readability, I have changed the offsets to reflect each stack variable's purpose.

```
mov      eax, [rbp+input_as_a_number]
mov      eax, eax
mov      edx, 0
div      [rbp+input_len]
mov      [rbp+input_as_a_number], eax
mov      eax, [rbp+input_as_a_number]
shr      eax, 2
mov      [rbp+input_as_a_number], eax
mov      eax, [rbp+input_as_a_number]
mov      edx, 0AAAAAAABh
mul      edx
mov      eax, edx
shr      eax, 1
cmp      rax, [rbp+input_len]
setnbe   al
```

Let us decode what is happening in this block of code. The input number is moved to eax. Then it is divided by the total length of the concatenated string. The result is stored in the input number variable, and a logical shift right 2 is performed on it. This value is then multiplied by the value 0xAAAAAAAB and shifted right 1. The calculated value is compared to the original input length and al is set to one if it is greater. To summarize the previous in a more readable format, in order to pass this stage your input must satisfy the following:

$$\frac{input\_number \div input\_len}{12} > input\_len$$

If so, the function returns non zero and the bomb continues to stage 6.

Pseudocode:

```
bool stage_3(char *user_input)
{
  int input_as_a_number;
  int input_len;
  unsigned __int64 v4;
  strncat(user_input, username, 128);
  input_len = strlen(user_input);
  sscanf(user_input, "%u", &input_as_a_number);
  return (input_as_a_number / input_len) / 12 > input_len;
}
```

Solve Script:

In order to solve the stage I wrote a script in Python 3 to quickly perform the math needed. The script is called "stage3.py" and is located in this repository for your convenience.

```
from os import getenv
username = getenv("USER")      # Calculate the length of the username
usernamelen = len(username)
ans = 0                        # Start at zero

# Loop until the first number is greater than
# the length of the number plus the username length

while True:
    ans += 1
    length = len(str(ans)) + usernamelen
    if (ans // length // 12) > length:
        break

print(ans)
```

On the UMBC server, this code gives me the integer value 1872.

5 Stage 4 Analysis

Analysis:

Stage 4's disassembly has conditionals and jumps, making a little bit more difficult to read but still straightforward.

```
mov      eax,DWORD PTR ds:0x201170        ; the username
movzx    eax, byte ptr [rax]
movsx    eax, al
mov      [rbp+first_letter_in_username], eax
```

stage_4 starts by saving the first character in the username in a local stack variable.

```
mov      r9, rsi
mov      r8, rcx
mov      rcx, rax
mov      rdx, r10
mov      esi, 0x40190a     ; "%u %u %u %u %u"
mov      eax, 0
call     sscanf
add      rsp, 10h
cmp      eax, 5
je       0x9
```

The binary then invokes the sscanf function again, this time with the format string "%u %u %u %u %u". This means that we need 5 integers to pass the stage. In fact, the binary checks if the function returns 5 and fails if not.

```
cmp      [rbp+counter], 4
jbe      0x401023
...
mov      rax, [rbp+counter]
mov      eax, [rbp+rax*4+input_numbers]
cmp      eax, [rbp+first_letter_in_username]
jbe      0x40103d
...
mov      rax, [rbp+counter]
mov      eax, [rbp+rax*4+input_numbers]
add      [rbp+first_letter_in_username], eax
jmp      0x401044
...
add      [rbp+counter], 1   ; continue back at top
```

This code shows the path of successful execution of stage_4. This can be thought of as a for loop that executes five times, over each number in our input. During each pass through the loop, the value is added to the ASCII value of the first character in the user name. If the next value is smaller than the previous, the stage fails.

```
mov      rdi, rax          ; username
call     strlen
mov      rcx, rax
mov      rax, rbx
mov      edx, 0
div      rcx
mov      rax, rdx
test     rax, rax
setz     al
```

After the loop, the stage calls strlen on the username and then divides the last sum by the length. If the remainder (in rdx) is 0, the stage returns non-zero. The pseudocode below shows an approximation of what is happening here:

Pseudocode:

```
for number in input_numbers:
  if number <= first_letter:
    return False
  else:
    first_letter += number
return first_letter % strlen(username) == 0
```

Solve Script:

I wrote the following script to solve this challenge:

```
from os import getenv
# Get the length of the username
username = getenv("USER")
length = len(username)
first_letter = ord(username[0])

nums = [first_letter + 1]

for i in range(4):
    nums.append(nums[-1] + (nums[-1] ))

# Add one if the last element is odd
if (nums[-1] % 2 == 0):
    nums[-1] += 1

print(" ".join(str(x) for x in nums))
```

This script gives me the following output on the server: 107 214 428 856 1713

6 Stage 5 Analysis

Analysis:

Stage 5 initially uses calls used many times in earlier stages, so I will skip over the minutiae. It calls sscanf on the user input to retrieve three values with the following format string: "%c %d %c". The stage expects a letter, followed by an integer, and another character. There is a call to strlen to get the length of the username.

```
mov     rax, [rbp+index]
cmp     rax, [rbp+username_len]
jb      0x40110a
```

This is the main loop of the stage and it terminates when the counter is not smaller than the username length.

```
mov    rax, QWORD PTR [rax*8+0x401928]
jmp    rax
```

This took a bit of research but these sources put me in the right direction:

- https://stackoverflow.com/questions/3012011/switch-case-assembly-level-code
- https://stackoverflow.com/questions/9815448/jmp-instruction-hex-code

The rest of the code represents a jump table based on the letters in the username. I look through the binary to find the offsets, and it is in .rodata. You can use the objdump tool to examine sections, so I run the following command:

```
objdump -s -j .rodata bomb.patched > objectdumps/.rodata
```

I write a python script where I annotate each jump for each value. It is called stage5.py. Once the code gets to the offset, one of the input characters is added to or subtracted from the input number. The calculation depends on the ASCII value of each character in the username. The actual calculations are in the stage5.py file.

Solve Script:

```
from os import getenv
username = getenv("USER")   # Get username
username_vals = [ord(character) for character in username]
input_num = 1000            # Just pick a high number. We will adjust later
master_input = 1000
char1 = ord('~')            # Use tildes because it's the highest printable ascii val (126)
char2 = ord('~')

def math_one(val):
    global input_num
    input_num -= (char1 + val)

def math_two(val):
    global input_num
    input_num -= (val * 2)

def math_three(val):
    global input_num
    input_num -= (val)

def math_four(val):
    global input_num
    input_num += char2 - val

```

```python
def math_five(val):
    global input_num
    input_num -= (val + char2)

def math_six(val):
    global input_num
    input_num -= (char2)

group_one = [97,101,105,111,117,121,math_one]
group_two = [98,99,100,math_two]
group_three = [102,103,104,math_three]
group_four = [106,107,108,109,110,math_four]
group_five = [112,113,114,115,116,math_five]
group_six = [118,119,120,122,math_six]

master_list = [group_one, group_two, group_three, group_four, group_five, group_six]

for value in username_vals:
    group = [x for x in master_list if value in x]      #find our list
    group[0][-1](value)

ans = master_input - input_num
print("{} {} {}".format(chr(char1), ans, chr(char2)))
```

This script gives me " 1172 " on the server.

7 Stage 6 Analysis

Analysis:

Stage 6

Solve Script:

8 Stage 7 Analysis

Analysis:

Solve Script:

9 Stage 8 Analysis

Analysis:

Solve Script:

10 Works Cited

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper.* (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`