Shellcode Analysis Lab

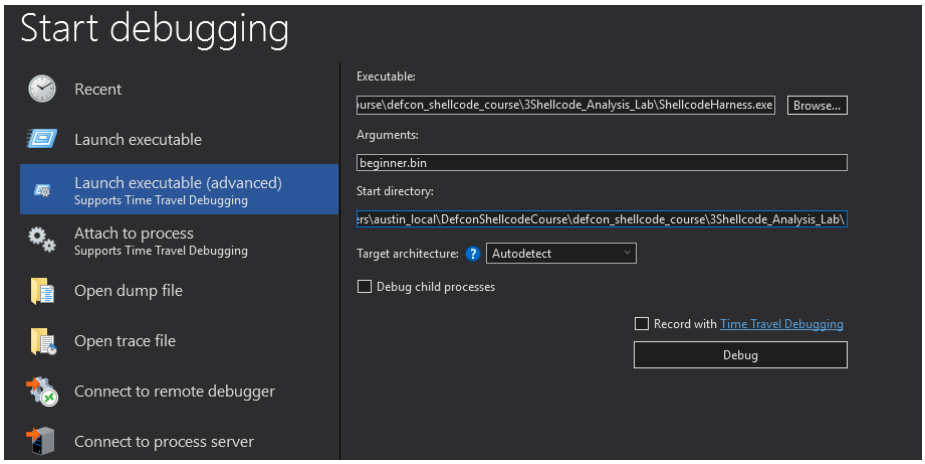
Installing SHAREM

- Make sure you have `git` installed on your Windows machine. Instructions to install git are located on the SHAREM Github page.
  - SHAREM Github
- Create python virtual environment so the dependencies do not alter your system python settings
  - `python -m venv sharem_venv`
- Activate the python virtual environment
  - `sharem_venv\Scripts\activate.bat`
- Clone the SHAREM repository
  - `git clone https://github.com/Bw3ll/sharem`
- Enter SHAREM directory
  - `cd sharem`
- Checkout a specific revision
  - `git checkout 091b9bff64cd8b04a49ae038dd3cefc422904da0`
- Install (part 1)
  - `python -m pip install .\sharem`
- Execute the Alternate Windows installer .bat
  - `cd .\installers`
  - `Windows_installer_alternate.bat`
- Confirm SHAREM is installed
  - `cd ..`
  - `python main.py`
  - You're looking for the usage menu

```
usage: Sharem [-h] (-pe PE | -r R | -r64 R64 | -r32 R32 | -d D) [-c C]
Sharem: error: one of the arguments -pe -r -r64 -r32 -d is required
```

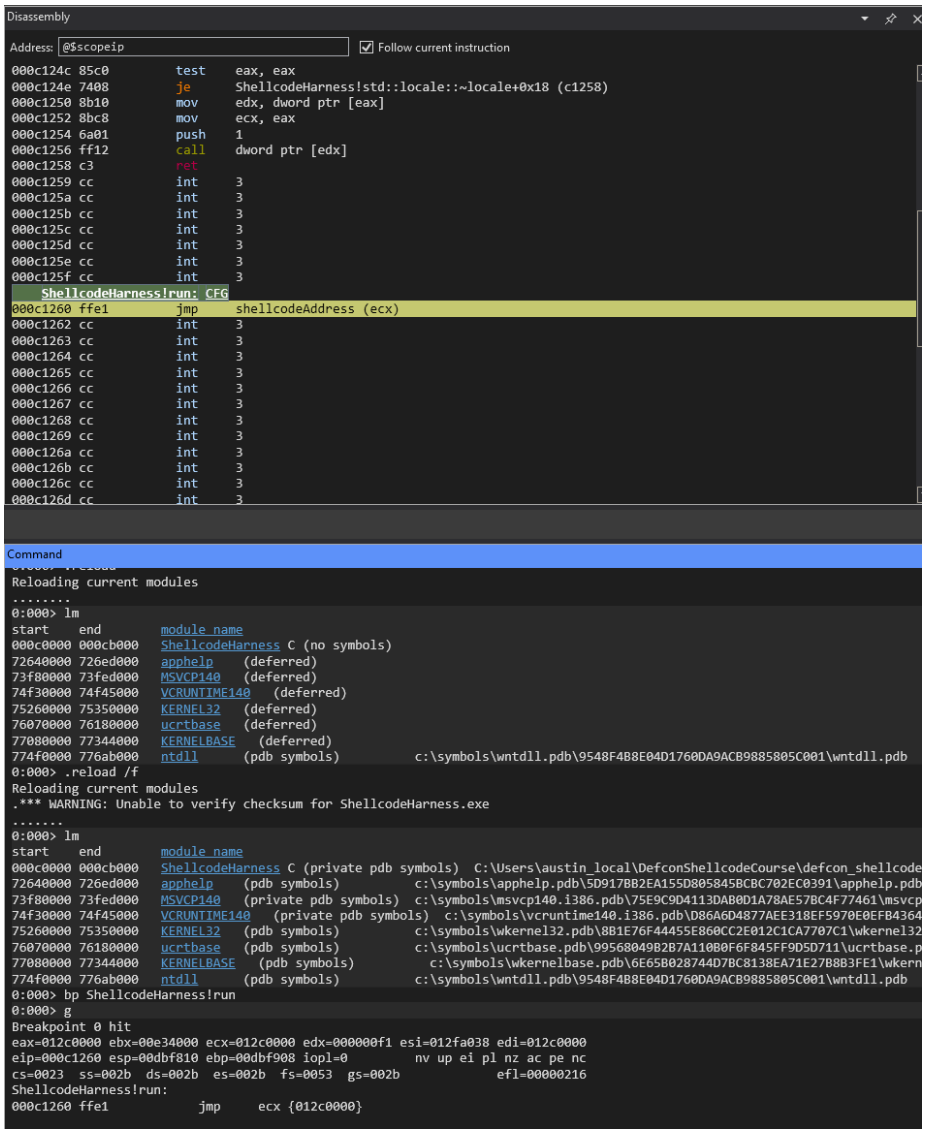
Opening Shellcode in a Debugger

0. If you do not have Python, NASM, or a debugger (WinDBG or x32dbg) installed, please do so now
1. Open up your debugger of choice, WinDBG or x32Dbg, with the target program being the shellcode harness
2. Run the Shellcode harness with the beginner32.bin



3. Execute `.reload /f` in Windbg to load the ShellcodeHarness.pdb file which gives symbols for the harness
4. Set a breakpoint on the run() function

- `bp ShellcodeHarness!run`
- When the breakpoint is successfully set, execute `g` to run the code until it hits the breakpoint.



5. Step through the execution and see if you can find the PEB, the dynamic resolution of kernel32.dll, and the dynamic resolution of two other functions in this shellcode sample.

- Often modules are page-aligned (ends with 000) and functions are 16-byte aligned (ends with 0)

Using SHAREM for Shellcode Analysis

1. Load the same .bin in Sharem and let it conduct it's analysis. How much faster did this get you the information you needed? Was all the same information accurate from your manual analysis with the debugger?
2. Use the `z` command to have SHAREM do a full analysis of beginner.bin
3. USe the `D` command to display the disassembly for beginner.bin
4. How much faster did this get you the information you needed? Was all the same information accurate from your manual analysis with the debugger? We will review after the lab to see if you were correct!

Repeat for Intermediate and Advanced Samples

1. Repeat these steps for intermediate and advanced shellcode samples and see how much easier using SHAREM can be than manually analyzing shellcode.
2. When you're finished, are there any cases where doing manual analysis may still be necesssary or useful?

Rolling Your Own Shellcode

Write a MessageBox Shellcode

Step 1

As you've seen from previous analysis, there are a lot of assembly instructions to get right before the shellcode can work. To start writing our MessageBox shellcode, we're going to make use of the `skeleton.asm` file. This file has a lot of details filled in but other parts that you will need to fill out.

The first task is that we need to know where we are in memory. To do that, we'll use the technique we learned to retrieve EIP or the Program Counter. When you get it, make sure to take notice of the actual value, it may be slightly off from what you need....

Hint: CALL places the **return** address on the stack that we `POP` into a register

Hint 2: When you're adjusting the value, make sure to account for the instructions that you are using to adjust the value. If you don't you'll run into a chicken-and-egg problem of every instruction to adjust the value causes the value to be wrong by a small amount.

Step 2

The next step is to get the address of the PEB. There is a hint in the skeleton.asm file that you can use to fill out the instruction.

Step 3

The next step will be figuring out the string comparison for your shellcode. This is one of the trickiest parts because you either need to write code to compare the bytes or compare hashes. Either option is fine based on what you want to do. We will discuss hashing a little later in this lab as well.

Step 4

Using your knowledge and ability to find information about windows internals, find out where the beginning of the data directory array is located in the PE file. You need to update an `ADD EAX, ???` instruction with this value to continue execution of your shellcode.

Step 5

Fill in the values needed to find the RVAs for the AddressOfNames, AddressOfNameOrdinals, and AddressOfFunction tables. There are hints in the skeleton to help you fill in these values.

Step 6

Implement the same, or another version, of string comparison for dynamically resolving GetProcAddress so we can use it to resolve other functions.

Step 7

Convert the string "ExitProcess" to a sequence of PUSH instructions with constant values. Remember, each character can be represented with a byte and Intel is a little-endian architecture.

Hashing Win API Function Names

Assembly Side

Now it's time to put that WinAPIHash.py script to good use. We can use this to generate hashes using our unique key and our function names to allow for Windows API look up without the function strings in our shellcode.

Here is the skeleton of the hashing function routine to use in your shellcode.

```
hash_func:
  ; Expecting values to be in ESI and EDI
  ; ESI - DWORD hash
  ; EDI - function name to be hashed
  ; ECX - Number of bytes
  ; Have ESP + 0, + 4, +8 to hold variables
  ; + 8 = original EDI pointer
  ; + 4 = loop count
  ; + 0 = temporary storage
  PUSH EBP
  MOV EBP, ESP
  SUB ESP, 0xC
  MOV [ESP+8], EDI  ; Store original value
  XOR EAX, EAX
  MOV [ESP+4], EAX  ; Set counter to 0
loop:
  MOV [ESP], EAX  ; Store EAX for a moment
  MOVZX EAX, byte [EDI]  ; Get byte into EAX
  TODO ; Add to previous value
  TODO ; Shift left
  TODO  ; Increment pointer
  TODO  ; Increment counter
  TODO  ; Check if we've reached the end
  JL loop
  XOR EAX, TODO  ; Unique Key
  TODO  ; Set return value, could be bool or DWORD hash
  MOV EDI, [ESP+8]  ; Restore original value
```

```
ADD ESP, 0xC
POP EBP
RET
```

You will have to fill in the gaps and use it within your shellcode to hash the function names and compare them in order to determine if you've found the right one. Make sure you pay attention to ASCII vs. Wide Character strings!

When you've completed the shellcode, assemble it using Nasm to create a .bin file.

Python Side

Now that you've completed making the encoder .bin file, copy these bytes into the your XOREncode_skeleton.py inside the XOR_DECODER_STUB global variable so it can be prepended to the encoded shellcode.

Now you have to fill in the rest of the Python XOR encoder script!

There are two main tasks to finishing this script:

1. XOR encode each of the bytes of the shellcode payload.
2. Include the size of the payload so the DECODER knows where to stop decoding.

For the first task, you'll need to lookup the python syntax for the XOR operation and implement it within your skeleton .py

For the second part, there is more flexibility. The solution to this part of the lab may change depending on how you implemented it in the xor_decode_skeleton.asm.

These instructions If you followed the instructions in the xor_decode_skeleton.asm, then there are three steps to implementing the size component for this encoder.

1. You need to get the size of the shellcode payload
2. You need to XOR it with whatever special key you're using in the xor_decode_skeleton.asm (by default it was 0xAABBCCDD)
3. You need to replace 0xFFEEDDCC with the encoded size in the XOR_DECODER_STUB array.

Ready To Go?

You should now have all the parts you need to run your encoded payload! Encode your payload and run it using the ShellcodeHarness and make sure it still works.

If your shellcode is not working, it's time to fire up the debugger and step through the execution to figure out what went wrong.

Creating Multi-WinAPI Shellcode

Lab

We are now going to build upon everything we've learned so far and transform our simple shellcode into a more complex, multi-Win API calling shellcode.

For this lab we are going to walk through setting up shellcode that will download a file, write it to disk, and then create a process from it. We will be using `URLDownloadToFile` and `CreateProcessA`.

URLDownloadToFile

Link to Docs

The first step will be modifying your shellcode to use `URLDownloadToFile`. This will require two steps:

1. Loading the urlmon.dll library into the process
2. Dynamically resolving the function

You have all the pieces you need to do both of these actions, remember you need to decide if you are using hashing or string comparison for the strings.

Once you have loaded the module and resolved the function you need to setup the call to `URLDownloadToFile`. You will need a URL and a file name; the rest of the parameters can be 0 in this case.

To simulate a web server, we will use python's built-in webserver:

```
python -m http.server 8080
```

Whatever working directory you run this from, copy calc.exe from C:\Windows\System32\calc.exe to this directory so it can be served by the python web server.

The URL that you will need to include in the shellcode is `http://127.0.0.1:8080/calc.exe`. This will be one of the parameters. The second parameter will be the location it's downloaded to. The directory structure must already exist, so we'll use one that we know exists, `C:\\Windows\\Temp\\`. We'll download the file as `boom.exe` to differentiate it from our `calc.exe`. Using this information, create the call to URLDownloadToFile in your shellcode.

CreateFileA

Link to Docs

Now we will need to create the process from the file we downloaded. Remember, the file should be located in C:\Windows\Temp\boom.exe.

There are many ways to setup the call to CreateProcessA, this lab will only walk you through one of them. First, we want to re-use the string for the file name

as the first argument to `CreateProcessA`. We will leave the second argument NULL, as `CreateFileA` can handle that and it's much easier for us to push a 0 on the stack than a string.

The third and fourth arguments will also be 0, followed by the fifth argument which is False. Since we are writing shellcode, there is no False so once again we will use 0 for bInheritHandles.

We will also specify 0 for the dwCreationFlag as this provides good defaults for us and ease-of-use.

Similarly, we will provide 0 for the environment pointer and the current directory pointer.

Unfortunately, that's where the ease of pushing 0s on the stack ends. We now need to create two Windows structures, `STARTUPINFO` and `PROCESS_INFORMATION`. These can be looked up from the docs link above. The good news is, while these structures are daunting for writing shellcode, only one field in both structures needs to be set and that is the `STARTUPINFO.cb` member needs to be set to the size of the `STARTUPINFO` struct.

Look up these structures on MSDN and use that information plus what you know so far to push pointers to these objects on the stack for the call to `CreateProcessA`.

Wrapping Up

Once you have managed to get our shellcode to work, demo it to the instructors by setting up your python server, copying calc to the correct directory, executing the shellcode in the handler, and showing that boom.exe (calc.exe) begins execution!

ShellWasp Lab

Setup

Clone the code from GitHub

- `git clone https://github.com/Bw3ll/ShellWasp`

Create a Python Virtual Environment and activate it

- `python -m venv .venv`
- `.venv\Scripts\activate.bat`

Install via Pip

- `pip install .`

Run ShellWasp

- `python shellWasp.py`

Lab Tasks

1. Start by simply creating a syscall shellcode, this will generate a huge file of shellcode.

- How does it get the PEB?
- How does it determine the version of the operating system?

2. Once the OS has been determined, we can see shellcode snippets setting up and calling the syscalls. The `ourSyscall` label ends up calling something else.

- What does the ourSyscall end up calling?
- Why is it necessary to call the syscall this way?