# Malware Development for Dummies

Cas van Cooten

x33fcon 2023
30-05-2023

# 00 | About

```
[cas@maldev ~]$ whoami
```

- Offensive Security Enthusiast, Red Team Operator, and hobbyist Malware Developer
- Likes building malware in Nim (but has recently been dabbling in Rust 🦀 )
- Author of tools such as Nimplant, Nimpackt, and BugBountyScanner
- Semi-pro shitposter on Twitter


Cas van Cooten

casvancooten.com

@chvancooten

chvancooten

/in/chvancooten

# 00 | About

About today's workshop

- Today we will be exploring the foundations of malware development!
- Exercises will throw you in the deep end (sorry not sorry)
- We will not have enough time to complete all exercises today! You are encouraged to keep practicing afterwards
- We will be targeting Microsoft Windows, but development can be done on any platform
- Slides, guidance on setting up a dev VM, exercises and annotated solutions are available at on Github:

**https://github.com/chvancooten/maldev-for-dummies**

# 01 | Malware Development

Why would a good guy do it?

- “Malicious Software”
- To defend against the bad guys, we should think like the bad guys (Sun Tzu quote here)
- Defenses are maturing, so we are forced to keep up
- In practice, malware can help us throughout various stages of the kill chain

# 01 | Malware Development

`Digital linguistics – choosing the right language for you`

- Many programming languages can be used, each with benefits and drawbacks
- Considerations:
  - High or low level
  - Interpreted or compiled
  - Developer experience (including docs)
  - Prevalence
- Support is provided for C#, Go, Nim or Rust, but feel free to choose whatever you are comfortable with! Discussed concepts are universal.

# 01 | Malware Development

## The MalDev Mindset

- Humble beginnings can be daunting
- Luckily, there is a great community of malware developers
- There are many excellent resources available that you can use as inspiration, cheat sheet, or even “borrow” some code from!
- Note: *Never* blindly copy-paste!
  - You don’t learn anything from it
  - Open-source tools are likely fingerprinted by defensive tools

Some great resources:

OffensiveNim

OffensiveCSharp

OffensiveRust

SharpSploit

OSEP-Code-Snippets

Dinjector

# 02 | Delivery

`Getting your payload executed`

- Payload delivery is critical for success
- For initial access, the payload type must be aligned with your pretext
- Every file type has opsec considerations
- Some examples:

**Binary executables**
.exe

**Dynamic link libraries**
.dll, .cpl

**Office files**
.xlsm, .doc, ...

**Office add-ins**
.xll, .wll, ...

**Shortcut files**
.lnk

# 02 | Delivery

`Getting your payload executed`

- Your choice of file type may impact your code (e.g. library versus binary versus script)
- Tools can be used to convert your malware to certain formats:
  - MacroPack
  - Donut
  - sRDI
  - …
- To keep things straightforward, we will stick with basic binary executables (.exe) today

# 02 | Delivery

```
We have execution! Now what?
```

- There are various execution techniques, each with their own behaviors:
  - Native functionality
  - Local shellcode execution
  - Remote shellcode execution (injection)
  - Reflective DLL injection
  - …
- For the exercises, we will be focusing on shellcode execution and injection only

# 03 | Shellcode Execution

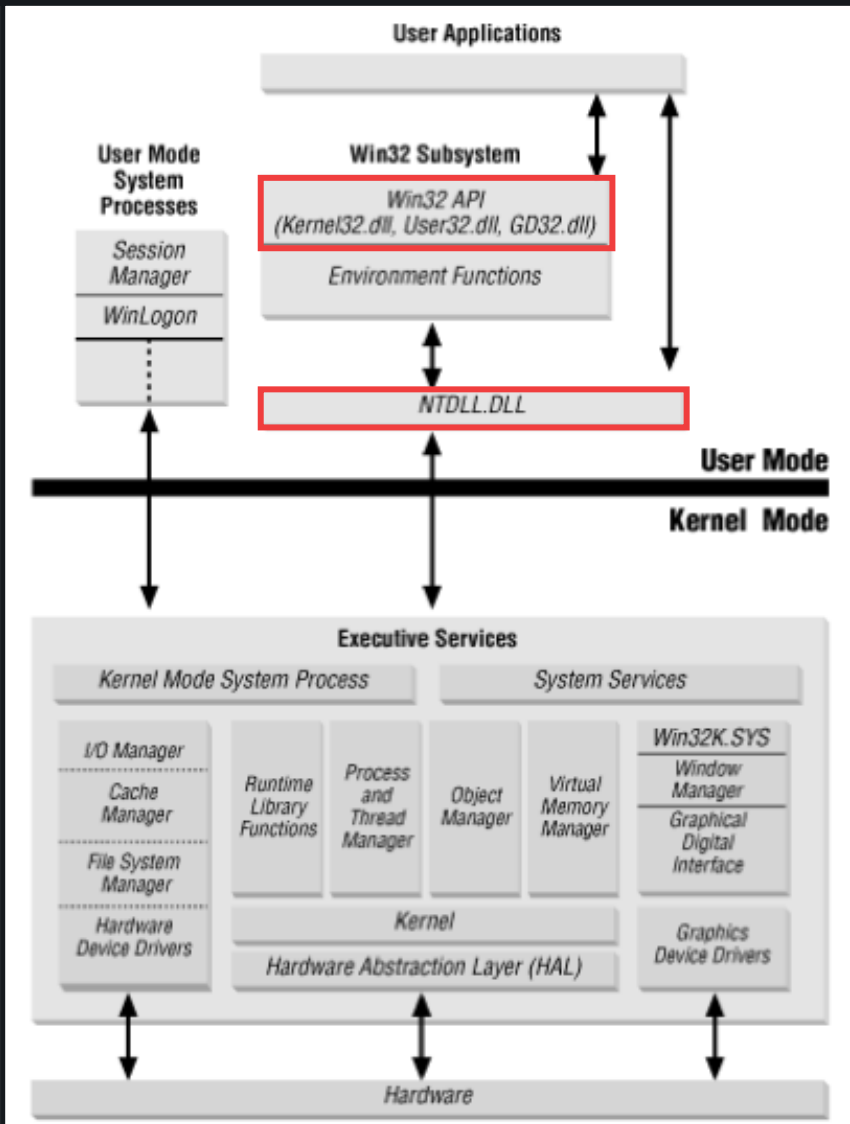
`Virtual-what now? Meet the Windows API`

- The Windows API is used to interface with all aspects of the OS
- Many functions available, can be used for all offensive use cases (enumeration to execution to lateral movement)
- (Mostly) documented on MSDN
- We will use it to load our shellcode 👀

# 03 | Shellcode Execution

## Win32 API versus Native API

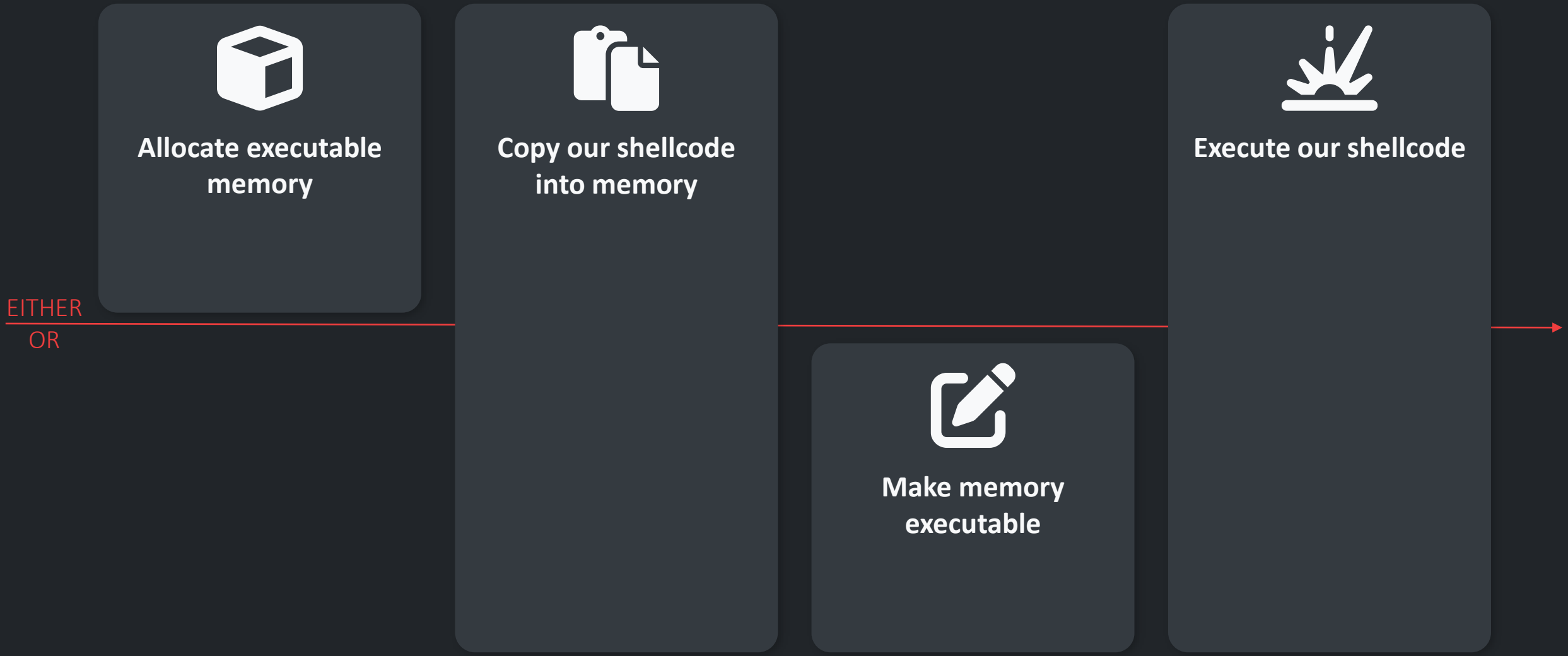
- There are various “levels” of API calls that you will encounter
- They do the same thing!
- Win32 API calls (such as `VirtualAlloc()`) are often just a wrapper for native API calls (such as `NtAllocateVirtualMemory()`)
- The Win32 API is easier to understand, but knowing the native API functions and their structure will help when looking at EDR evasion later




Source: MSDN

# 03 | Shellcode Execution

## Shellcode execution techniques



# 03 | Shellcode Execution

## Shellcode execution techniques

**Allocate executable memory**

`VirtualAlloc()`
`NtAllocateVirtualMemory()`

EITHER

OR

**Copy our shellcode into memory**

`RtlMoveMemory()`

Or use native functionality exposed by a language, such as:
`Marshal.Copy (C#)`
`copyMem (Nim)`
`std::ptr::copy (Rust)`

**Make memory executable**

`VirtualProtect()`
`NtProtectVirtualMemory()`

**Execute our shellcode**

`CreateThread()`

# Exercise_

## Build a basic shellcode runner

00 | Set up a dev VM with tools for your chosen language

01 | Use `msfvenom` to generate some shellcode, and write a basic loader that executes it in the current process

B01 | Modify your loader so that it executes shellcode without calling `CreateThread()`

Guidance on GitHub

# 04 | Shellcode Injection

```
Execution in a remote process, don’t mind if I do
```

- Shellcode execution in another process
- Injection is opsec-expensive, but malware running in the context of an existing process can have great benefits!
- We need a handle to operate in another process
- We can only get a handle on processes we have permissions for (typically current user context)
- If we’re not sure a process exists, why not spawn it?

# 04 | Shellcode Injection

`New API calls for injection`

- We can use a similar allocate-write-execute approach
- Getting a handle:
  - `OpenProcess()` or `NtOpenProcess()`
  - Afterwards, clean up with `CloseHandle()` or `NtClose()`
- Allocation:
  - `VirtualAllocEx()` or again `NtAllocateVirtualMemory()`
- Copying:
  - We *need* to use the Windows API this time, since we're dealing with handles
  - `WriteProcessMemory()` or `NtWriteVirtualMemory()`
- Execution:
  - `CreateRemoteThread()` or `NtCreateThreadEx()`

Why not get creative with your API calls?

MalApi.io has a neat list of API functions that can be abused.

# Exercise_

## Build a basic shellcode injector

02 | Create a new project that injects your shellcode in a remote process, such as `explorer.exe`

B02 | Make the target process configurable, and spawn the process if it does not exist already

Guidance on GitHub

# 05 | Defense Evasion

`Bypassing defenses like the big boys`

## In a real scenario, you are up against many layers of defenses

### Antivirus (AV)

- The most basic defense, but not to be underestimated
- Mostly looks at files statically
- Sometimes uses a sandbox to inspect basic behavior
- Blocks shady stuff

### Enterprise Detection and Response (EDR)

- AV on steroids
- Usually uses advanced behavioral detections
- ‘Hooks’ APIs and scans memory for indicators
- Does not always block, may ‘only’ alert!

### The Blue Team

- One alert can be enough to ruin your operation
- May dissect your malware to find out more about you
- Will ruin your day

### ... many others

- Threat hunters
- Other endpoint-based controls
- Network-based controls
- Behavioral analytics
- ...

# 05 | Defense Evasion

`Defensive decision-making`

## Evasion is effectively a combination of the below (and a bit of luck)

**Avoid**

- Avoiding locations or activities that are under defensive scrutiny
- E.g. proxying tools rather than executing on a victim endpoint

**Blend In**

- Making telemetry generated by your malware look as legitimate as possible
- Also involves making clever use of defensive ‘blind spots’!

**Sabotage**

- Tampering to disrupt the data flow used for defensive purposes
- E.g. patching AMSI/ETW or unhooking function calls

# 05 | Defense Evasion

AV evasion

- AV evasion is relatively simple, getting rid of “known bad” indicators is usually enough
- **Obfuscation** can help get rid of suspicious indicators
- Strings, shellcode, and function calls can all be obfuscated (automatically)
- Encryption or encoding (even just XOR or ROT) of shellcode is a bare minimum
- Too much obfuscation is an indicator in itself 👀

You can test your evasions using something like ThreatCheck

Be **very** careful with the submission of payloads to VirusTotal, as defenders automatically ingest and analyze these payloads

Antiscan promises to not do this, but there are no guarantees...

# 05 | Defense Evasion

`More AV evasion`

- Logic bypasses
  - AV takes shortcuts to minimize resource use, we can abuse these!
- Sandbox evasion
  - Perform benign calculations for 30-60s
  - Check for devices, resolution, user input, etc.
- Payload keying
  - Ensuring payload will only fire in target environment
  - Often by using target environment (e.g. domain name) as encryption key

Some inspiration:

Evasions

CheckPlease

KeyRing

DripLoader

ConfuserEx

Denim

# 05 | Defense Evasion

`EDR evasion`

- EDR uses a variety of telemetry sources (API hooks, kernel callbacks, ETW, ...)
- Prioritize blending in over tampering
- Some popular bypass methods:
  - (In)Direct system calls
  - API unhooking
  - Sleep masking
  - Stack spoofing
  - ...
- EDR evasion is tough, don't expect to nail it first try
- Remember: No block != no alert

Further reading:

"Blinding EDR on Windows"

"A tale of EDR bypass methods"

"Let's create an EDR... And bypass it!"

# Exercise_

## Make your malware evasive

03 | Implement one or more of the described evasion techniques in your shellcode loader / injector and test it against AV

B03 | Implement one or more of the mentioned EDR evasion techniques (test it against EDR if you are able)

Guidance on GitHub