Packing Heat!

Dimitrios A. Glynos
{ dimitris *at* census-labs.com }

Census, Inc.

AthCon 2012 / Athens, Greece

Overview

Introduction

Evading detection from AV software

Producing metamorphic executables

Implementing a metamorphic packer

Evaluation

Conclusions

INTRODUCTION

based on photo by Thomas Angermann

Executable packing

- A (runtime) packer is a piece of software that places an application (and sometimes other related files) inside an executable container file
- At execution time the container loads and executes the “packed” software (payload)
- A packer may compress and/or encrypt the container contents

Executable packing

- Why use a packer?
  - To decrease on-disk application size
  - To hide application internals
  - To enable the execution of pentest (or other malicious) apps on hosts protected by AntiViruses (AV) or IPS
- In this presentation we’ll focus on PE packing for AV evasion purposes

AntiVirus software

- Originally, a means for disinfecting systems from software viruses
- Nowadays, they also protect hosts from other types of malicious software activity
- Poor man’s HIPS

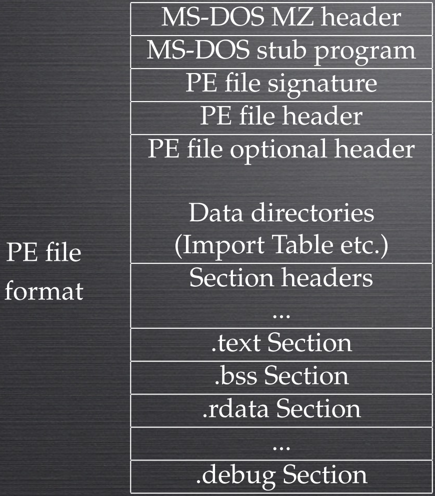
AntiVirus software

- Automatic malware detection based on
  - Static analysis (signatures, imports, etc.)
  - Dynamic analysis (suspicious calls, heuristics etc.)
- Two main modes of operation
  - Identifying malware at scan-time
  - Identifying malware at runtime
- Malware classification is a non-trivial process

EVADING DETECTION FROM AV SOFTWARE

based on photo by foqus

Evading static analysis techniques



PE file format

MS-DOS MZ header

MS-DOS stub program

PE file signature

PE file header

PE file optional header

Data directories (Import Table etc.)

Section headers

...

.text Section

.bss Section

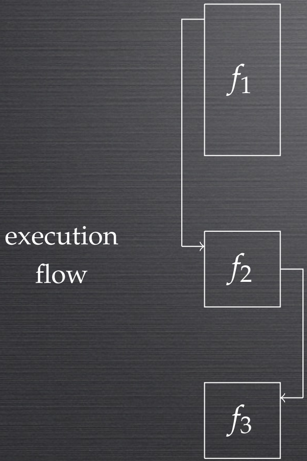
.rdata Section

...

.debug Section

- Encode payload
- Generate a different PE each time
- Normal PE structure
- No signatures from section/header data
- Keep entropy low
- Standard MS-DOS stub
- Refer to an unsuspicious set of external functions
- Unique Control Flow Graph

Evading dynamic analysis techniques



- Model the behavior of innocent apps
- Load code and data at different memory locations for each PE
- Make each PE have a unique Call Graph
- Handle tracing / emulation
- Take per AV measures
- ...
- Pray that the AV will give up before spotting the embedded known malware

Producing metamorphic executables

A typical packing scenario

PE Container data

PE stub data	Allocator	Decoder	Loader	Enc. Payload	...

- At build time, the packer
  - **Encodes** (compresses, encrypts etc.) the payload
  - **Installs** the payload in a section of a “stub” PE file
- At runtime, the container
  - **Allocates** memory
  - **Decodes** the payload (in the allocated memory)
  - **Loads** (and executes) the payload

Problems with this design

PE Container data

PE stub data	Allocator	Decoder	Loader	Enc. Payload	...

- The packer output is immediately identifiable
    - Pieces of the stub can be used as a signature
    - The Allocator, Decoder and Loader code can also be used as a signature
- What’s the problem with identifying the packer?
    - If the loading process is always the same, the AV knows *when* loading has finished
    - It can wait until then to extract and analyze the original payload

Two helpful techniques

- Polymorphic Encoding
  - Encrypt code with random key
  - Instructions will be decrypted and executed at runtime
- Metamorphic Encoding
  - Reimplement a set of operations with equivalent instructions
  - Special software generates the equivalent code automatically

A better packer design

PE Container data

PE data	Met. Allocator	Met. Decoder	Enc. Loader	Enc. Payload	...

- At build time, the packer
  - **Generates** a new metamorphic Allocator and Decoder
  - **Encodes** the Payload and Loader (polymorphic encoding)
  - **Incorporates** all components into a new container
- At runtime, the container
  - **Allocates** memory
  - **Decodes** the Loader and Payload
  - **Loads** and executes the Payload

ONLY PROBLEM IS…

Need a way to

- generate the metamorphic code *on-the-fly*
- create the PE container from scratch
- integrate all components seamlessly
- have full control over the output of each phase
  - necessary for fooling static/dynamic analysers

Implementing a metamorphic packer

METASM - THE RIGHT TOOL FOR THE JOB

- A Ruby framework that provides *"a cross-architecture assembler, disassembler, compiler, linker and debugger"*
    - See [METASM]
- Idea:
    - Make the packer a Ruby script!
    - Develop a library of metamorphic instructions
    - Implement the Allocator, Decryptor and Loader using these instructions
    - Assemble with METASM
    - Encrypt Payload and Loader bytes in Ruby
    - Link intermediate object code using METASM
    - Generate final PE file using METASM
- Ruby + METASM make our packer cross-platform!

Step 1: Become familiar with METASM

```
pe = Metasm::PE.assemble Metasm::Ia32.new, <<EOS
.entrypoint
push 0
push title
push message
push 0
call messagebox
xor eax, eax
ret
.import ’user32’ MessageBoxA messagebox
.data
message db ’Hello World!’, 0
title   db ’Messabox Title’, 0
EOS
pe.encode_file ’output.exe’
```

- Script-level assembler control: a powerful tool!
  - Dynamic selection of registers, instructions etc.
  - Dynamic creation of symbols, labels etc.

Step 2: Develop a metamorphic instruction library

```
def self.add_reg_dword(reg, val, _avoid_regs=[])
        avoid_regs = Array.new(_avoid_regs)
        avoid_regs << reg

        methods = [
                Proc.new {
                        "add %s, %i\n" % [reg, val]
                },
                ...
                provide other alternative implementations here
                avoiding the use of protected registers found
                in "avoid_regs"
                ...
        ]
        method = methods[ rand(methods.length) ]
        return method.call()
end
```

- Keep this private!

STEP 3: ENCODE VALUES

- Hide particular constants by doing arithmetic with random numbers
- Strings can be encoded in a similar fashion ;-)

Step 4: Implement WinAPI wrappers

- Create wrappers for useful WinAPI functions
- Resolve a function's address via `GetProcAddress`
- Use metamorphic instructions to place the function's arguments on the stack
- Execute the function via a metamorphic "call" instruction

Step 5: Implement the Allocator

- Use the WinAPI wrappers to build your memory allocator
- It’s good to use memory blocks that do not always start at the same memory address

Step 6: Implement the Decoder

- Decide on a payload encryption method
- Pick a random key
- Insert the key at a random place in the PE file
- Prepare a metamorphic decryptor
- Decrypt inside the previously allocated memory blocks
- Key may also be derived from the execution environment or other context
    - See our Context-keyed Payload Encoding [CN10] presentation from AthCon 2010

Step 7: Implement the Loader

- Pick your favorite loading technique
  - For an example, see [PELOAD]
- Use the WinAPI wrappers to implement the loader
  - This makes the loader metamorphic too!

Step 8: Be creative!

- Introduce garbage...
  - Garbage instructions (like no-ops)
  - Garbage calls
  - Interpolate real code with garbage
- Play games with the execution flow
  - Introduce conditional branches
- Randomize the execution flow
  - Create a dependency graph of code components
  - At build time, randomize placement of components
  - Runtime equivalent – use a dispatcher
- Create a seemingly innocent import table
  - Don’t just put functions there, use them too!
- Insert (metamorphic) anti-emulation code

Step 9: Implement the PE generator

- Assemble all intermediate components
- Encrypt Loader and Payload
- Link all components together and generate the PE
- Make the resulting PE look standard
    - Examine a standard Windows application
    - Check header values
    - Check section attributes
    - Modify the container structure accordingly

TRIVIA

- Prototype implementation: 1700 lines of code
    - A total of 24 polymorphic instructions and WinAPI wrappers
- Did (almost) all of the development under Linux with the assistance of winedbg
- Found Ruby to be a bit slow at encryption / shuffling (but that could be my fault)

Evaluation

VirusTotal statistics

- Test payload
  - Metasploit “TCP reverse shell” for Windows
- Without packing
  - Detection ratio: 33/42
- With packing
  - Detection ratio: 6/42
  - 5 warnings about suspicious / packed file
    - Same 5 warnings for packed innocent file
  - 1 AV flagged this as a trojan
- Complete stealthiness is tricky
  - Prepare yourself for some long hours of fine tuning!

DEMO

Conclusions

Notes on detecting malicious executables

- Employ both static and dynamic analysis techniques
- Signature matching will always come in handy
  - Fastest method of detection
- Strive to detect the payload, not just the packer
- See [CS10] by Silvio Cesare for ways to detect when unpacking has finished
  - Useful for detecting known malicious payloads
- At runtime, identify *groups* of calls to library functions (and system calls) that are indicative of malicious behavior
  - Useful for detecting both known and unknown malicious payloads

Concluding remarks

- Presented a novel design for a metamorphic packer
- METASM provides a cross-platform toolchain for building such packers
- Example implementation produces malicious executables that evade detection from a large number of AV software
- As metamorphic malware becomes the norm, AV vendors must invest on better runtime analysis techniques
- AVs are no substitute for user awareness!

References

- Microsoft PE and COFF Specification
  `http://msdn.microsoft.com/windows/hardware/gg463119`
- Portable Executable Loaders and Wrappers
  `http://www.cultdeadcow.com/tools/pewrap.html`
- The METASM assembly manipulation suite
  `http://code.google.com/metasm`
- Context-keyed Payload Encoding: Fighting the Next Generation of IDS
  by D. A. Glynos, Census Inc., AthCon 2010
- VirusTotal - Free online Virus, Malware and URL Scanner
  `http://www.virustotal.com`
- Fast Automated Unpacking and Classification of Malware
  by Silvio Cesare, Master's Thesis, 2010

QUESTIONS?