Esteganography and binary obfuscation

David Guillen Fandos
Tarragona – Spain

david@davidgf.net
davidgf.net – github.com/davidgfnet

Steganography

- What is steganograpy?

... is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message

(from Wikipedia)

Steganography

In brief:

Hiding a message so it passes unnoticed

What steganography is not:

- Encryption
- A secure form to send/store data

This is the so called 'security through obscurity'

Steganography examples

- Ancient China, Greece and Egypt
  - Send messages hidden in human body
- Messages hidden in popular writings:
  - Bible code
- Modern implementations:
  - Invisible ink
  - Many examples during 2nd World War

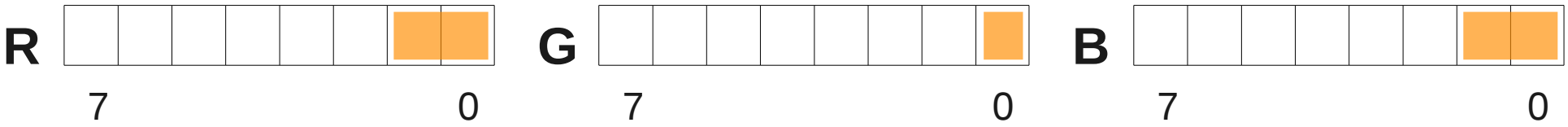
Modern steganography

- New techniques
  - Digital images
  - Audio (digital and analog)
  - Video
  - Rich/plain text (like HTML)
- Advanced techniques
  - Any digital file (or almost)
  - Protocols (network, APIs...)

Image steganography

Idea: use low significant pixel bits to encode data

- Each pixels is encoded using 3 integers (RGB)
- Slightly change the values to add our data!



- Use uncompressed image formats (PNG, BMP, ...)

Result: Added noise to color data (unnoticeable)

Audio steganography

Use low significant sample bits to encode data

- Digital audio uses LSB of samples (uncompressed)

(DIGITAL AUDIO)

Add low power signal over the original one

- Noise may corrupt hidden data

Use higher/lower frequency bands

- Not audible bands
- Channel may corrupt hidden message

(ANALOG AUDIO)

Redundant encodings

It is possible (and very common in digital formats) to encode the same information in different ways.

- Sometimes it's inherent to the channel
  - Instructions in a program
  - Messages on networks
- Or used as error protection
  - Redundant encoding such as ECC, Reed-Solomon
  - Very popular in QR codes (to embed a logo)

Detection

How can we detect hidden information?

- Tipically statistical analsysis are used
  - For LSB image steganography it's easy to detect correlated bits when a message is present
  - If the message is pseudorandom is not that easy!
- In general is non-trivial to detect hidden messages

x86 encoding

Those concepts also apply for executables

- Programs may be implemented in many ways
- High expressivity at assembly level
  - Specially on CISC architectures
  - x86 has tons of legacy instructions
  - And many weird encodings ...

Exploit this!

Hydan

Hydan implements code steganography

- Embeds hidden messages in x86 executables
- Exploits some peculiarities of x86 instructions:
  - Multiple instruction encodings
  - Different instructions for the same operation
  - Instruction reordering

`http://crazyboy.com/hydan/`

Hydan

Examples:

add eax, 1 → sub eax, -1

xor eax, eax → sub eax, eax

lea eax, (ebx,ecx,1) → lea eax, (ecx,ebx,1)

Hydan issues

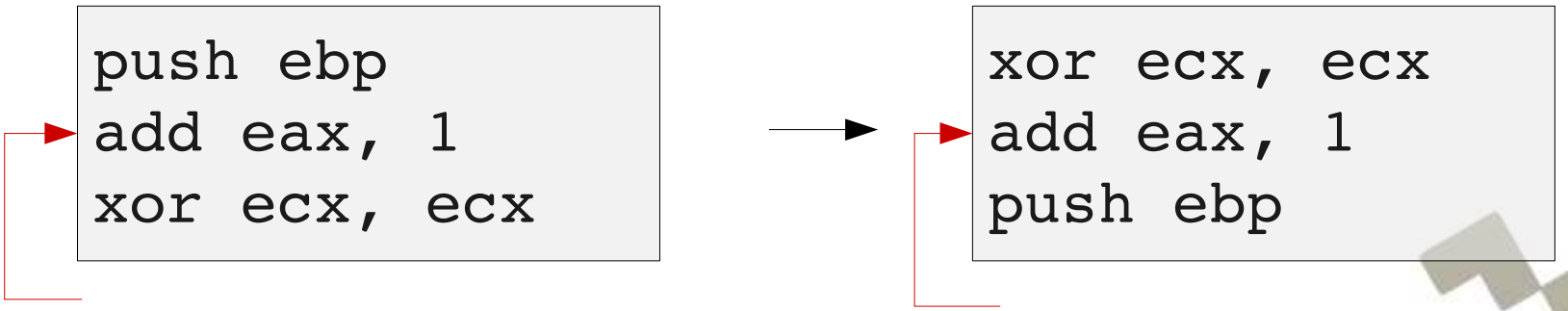
Technique used by Hydan has some issues

- Low encoding ratio
  - To be fair, it depends on the executable
  - Not many embdeddable instructions
  - Instr. reordering is difficult without compiler info.
- Complicated algorithms to embed data and extract it
  - It is desirable to have easy data extraction

Binary patching

Modifying existing binaries is difficult

- Cannot perform relocations (or really hard)
  - Needed for instruction reordering
  - Also for instruction replacement (different size)
  - Replacing blocks of instructions is great but hard



3! possible permutations ~ 2.58 bits

Hydan detection

Hydan could be detected under some circumstances

- Matching typical compiler generated blocks
  - For example, prologue may always do 'sub esp, 8' and Hydan may replace it by 'add esp, -8', which determinant.
- Opcode distribution variation
- Instruction reordering detection

Idea!!!

Instead of using an existing executable create a new one to embed data within!

Why?

- Executable code under our control
  - We may choose instruction types
  - Control executable size
- Generation is straightforward (more or less...)
- Extraction is also straightforward (or can be)

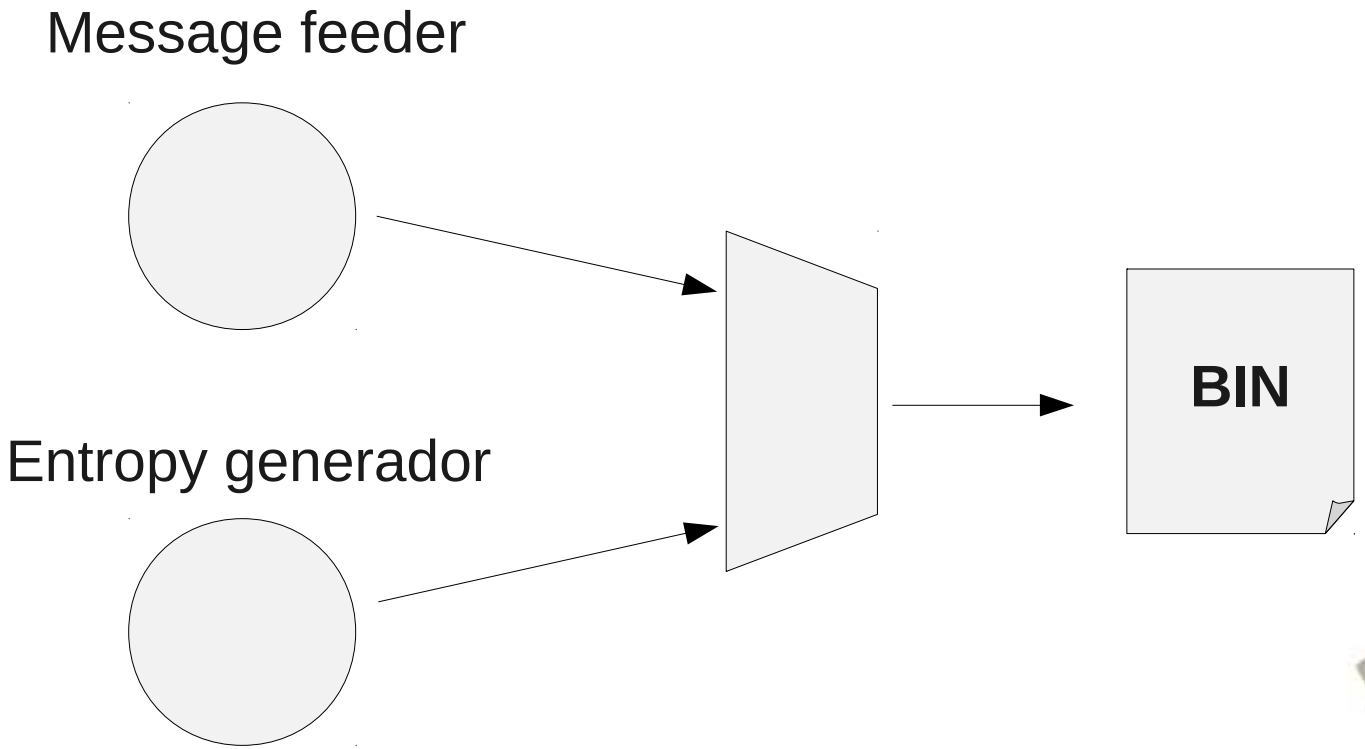
Custom code generation

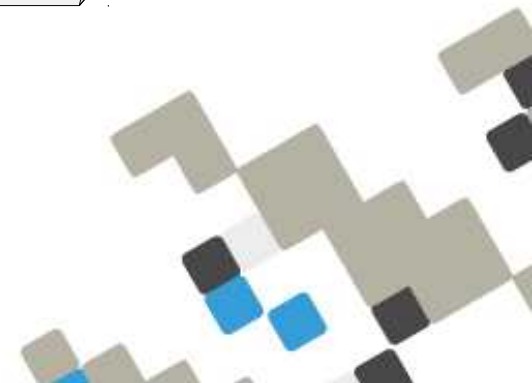
Create code similar to compiler generated binary

- Correct codestream
- Functionally correct!?
  - We may even execute it!
- Compiler-style code
  - Function blocks
  - Proper prologues and epilogues

Custom code generation

Code generation with embedded message

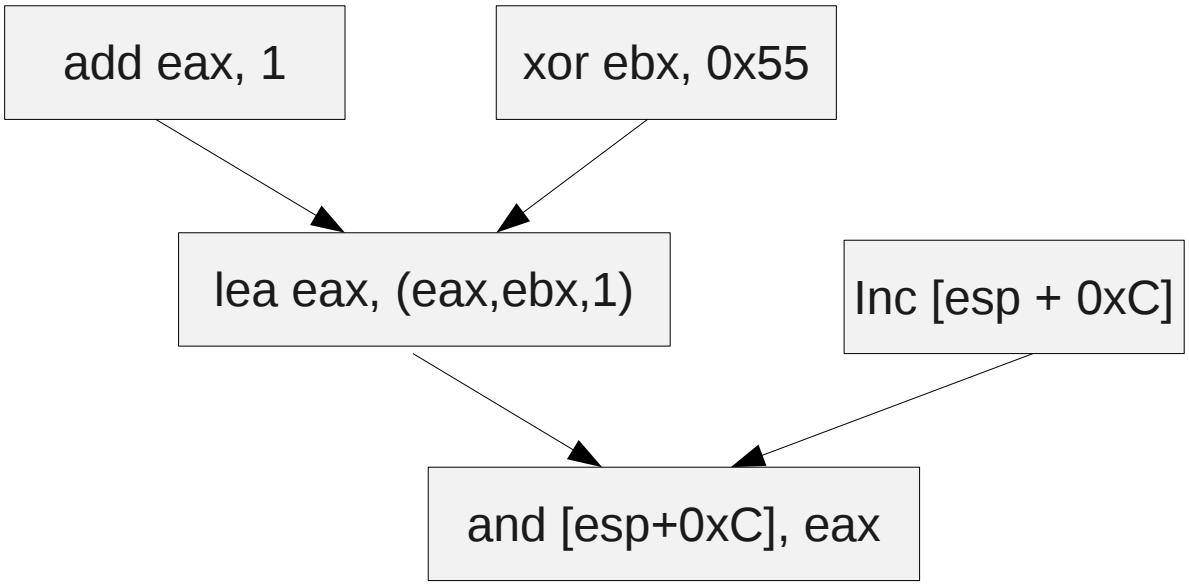




Code properties

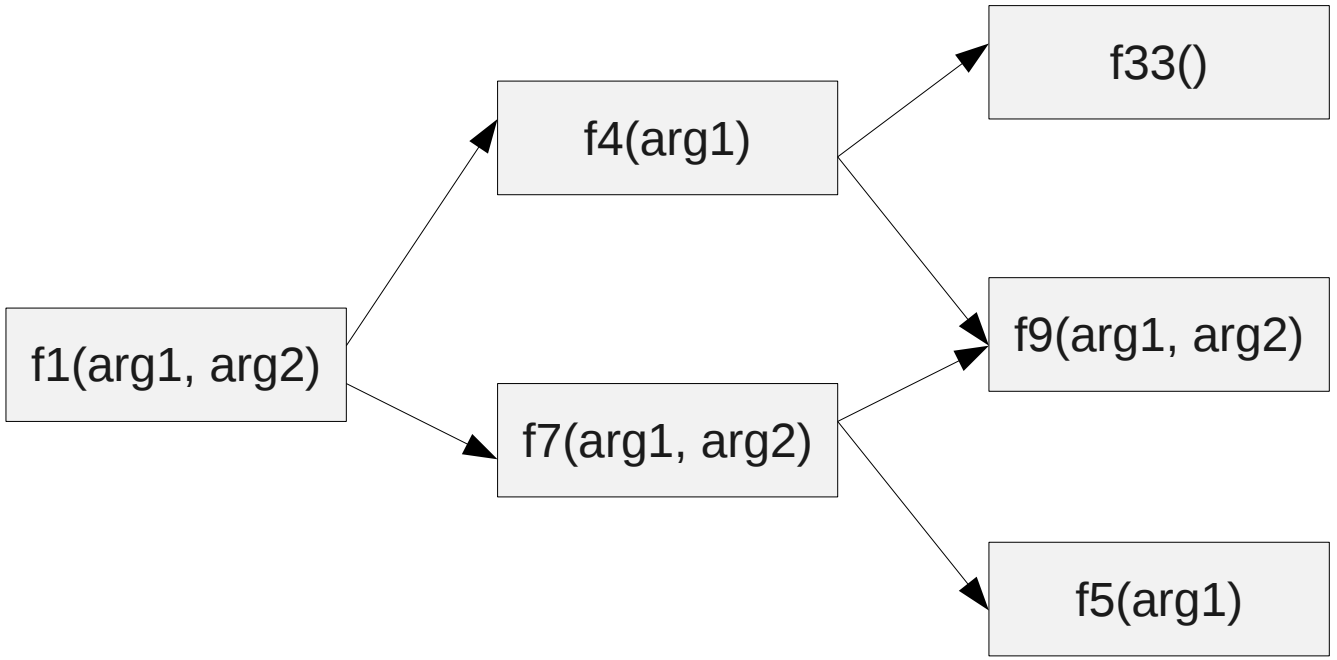
In order to have runnable code we have to guarantee:

- Initialized data access (mem & reg)



Code properties

- Finite loops & function recursion



Code properties

- Indirect variables
  - As complex as we desire
  - Create struct/array objects
- Data may come from
  - Parameters
  - Global variables
  - Local variables
  - Registers

Show me the stuff

Example!

Transform a simple TXT file into x86 bytestream

So ... what?

Nice demo, right? Now what?

What is the purpose of binary obfuscation, what can we do with it?

You know...

Crypters!

We can create a nice crypter with this!

What How Why? Let's start from the beggining:

”A crypter is a program which encrypts executables to make them undetectable from AntiVirus software”

Some internet random site

Notes on AVs

How AVs work? Many different types of analysis

- Static analysis
  - Takes a look at the binary using algorithms
  - Typically using a lot of pattern matching (signature)
- Dynamic analysis
  - Runs (actually emulates) the executable to see how it behaves
  - It's not a real execution, it emulates OS calls

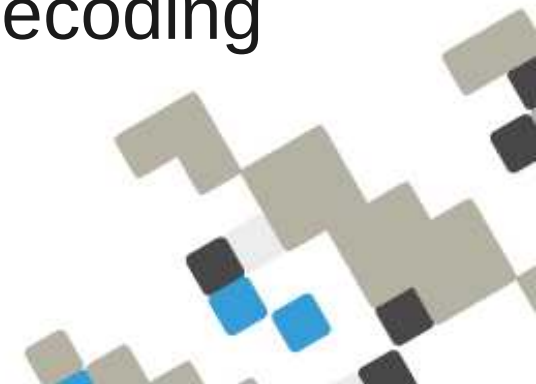
Notes on AVs

Why it's good to use x86 steganography?

Because we can make static analysis really difficult:

- The data is really really obfuscated
- Also we can easily mutate our malware
  - It's easy to decode and reencode the data
  - It's even possible to mutate it without decoding
- Really interesting feature:

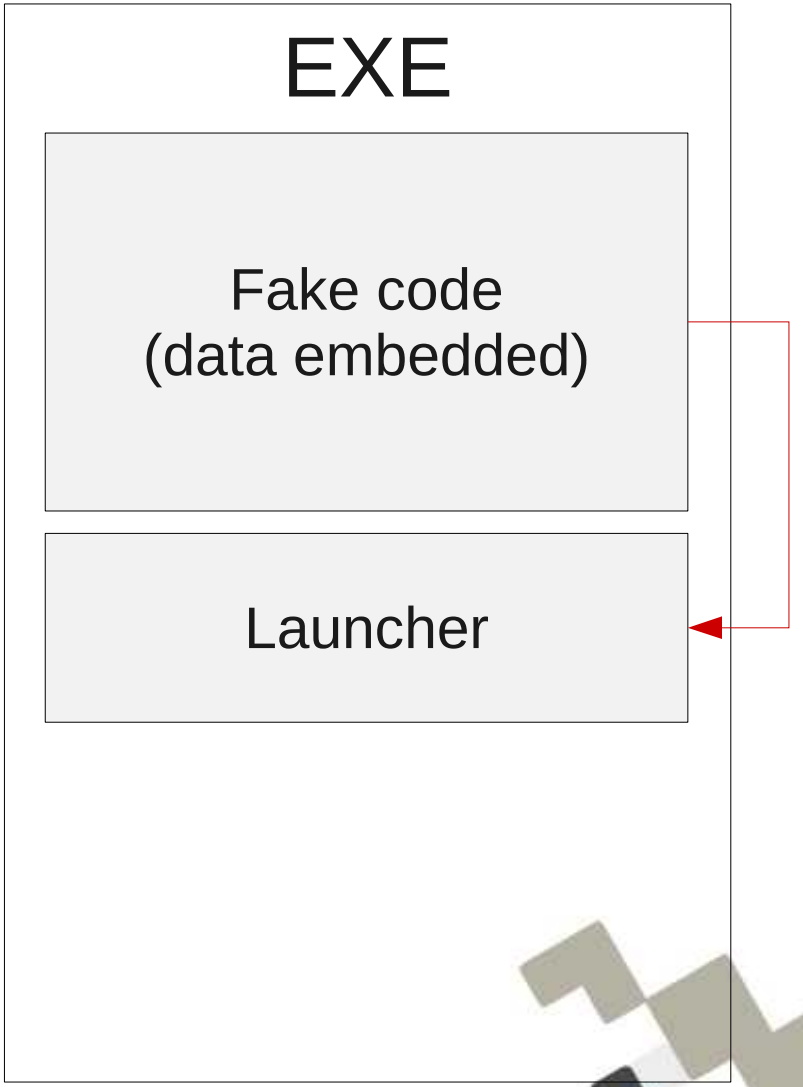
  We can ”run” our data!



How the crypter works

Generates an EXE with all the code plus a launcher called 'stub'

- The fake code is executed first
  - This discourages AV from emulating it all
- Then the launcher runs the original executable



How the crypter works

We work under the premise that the AV will never think that code itself contains hidden data...

Because he is executing it!

But... pitfalls:

- Dynamic analysis will reveal the purpose of the executable, only good for static analysis
- We still have the launcher, which can be statically analyzed

Let's watch the thing runin'

Demo!

Encode an EXE, then create a new EXE with the launcher and the data.

Conclusions

We found a good way to store data within an executable which is not suspicious compared to other techniques.

It is **just** good for static analysis, not dynamic

Needs to be used along with some dynamic techniques to confuse AVs.

We are not finished! Many things to do!

This is the end

Thank you!

Questions?