Implementation of Runtime PE-Crypter

BerlinSides 0x3

Christian Ammann

Content

- Introduction
- Portable Executables
- Windows PE-Loader
- Implementation of Runtime PE-Crypter
- Demonstration
- Further Work

Problem: Deployment of Malware

Victims PC

Your Malware

Anti-Virus Software

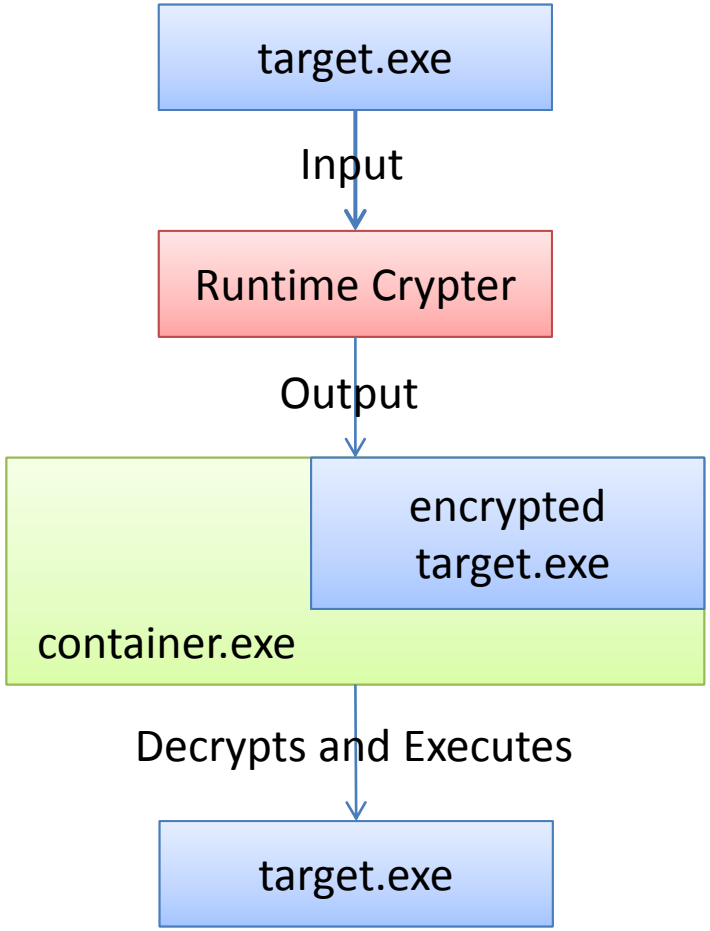
Idea: Encryption of Executable

Malware before Encryption

Malware after Encryption

Anti-Virus Software

Workflow of a Crypter



Use Cases

- Encryption of Malware to bypass AV protection
- Obfuscation of binaries as a protection against reverse engineering
- Remove the encryption and add a compression algorithm to reduce binaries file size
- Merge two binaries

Problem: Obtaining a PE-Crypter

Deal with mature people like this

Problem: Obtaining a PE-Crypter

- Use a free PE-Crypter
  - Often detected by AV scanner
- Use a commercial PE-Crypter
  - Can be expensive
  - No control of the code base

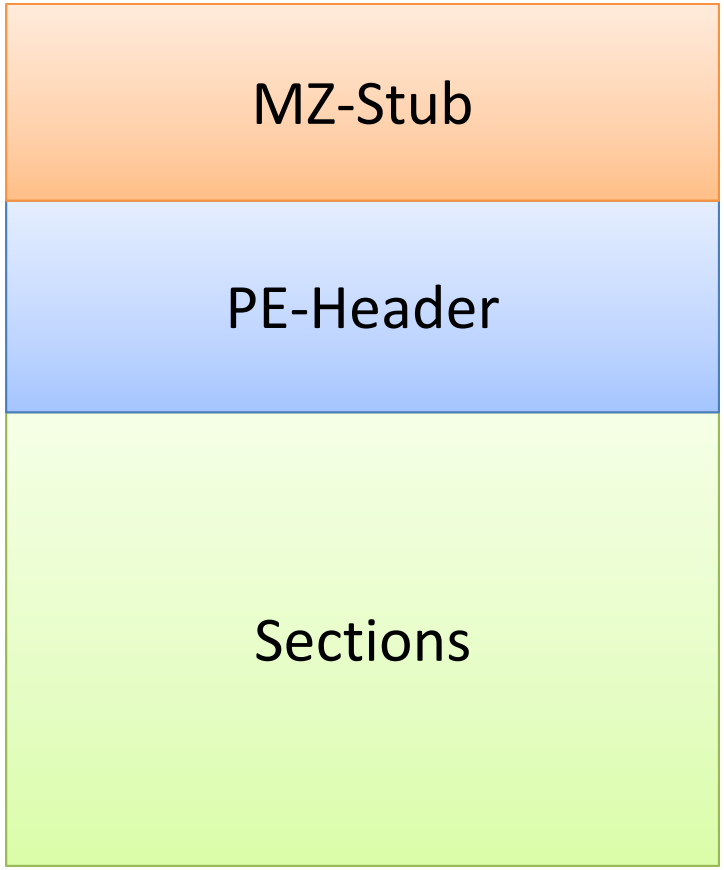
Solution

Develope your own PE-Crypter

Portable Executables

- Windows binary format for executable files
- Two categories: Image- and Object-Files
- Used in 32-bit and 64-bit systems
- DLL‘s are also Portable Executables:
  - Contain a list of functions which can be called by other Executables
  - Can be dynamically loaded with LoadLibrary() and GetProcAddress()
  - Example: user32.dll exports MessageBoxA() and allows the userland to create a message box

Portable Executable Format



Portable Executable Format in Detail

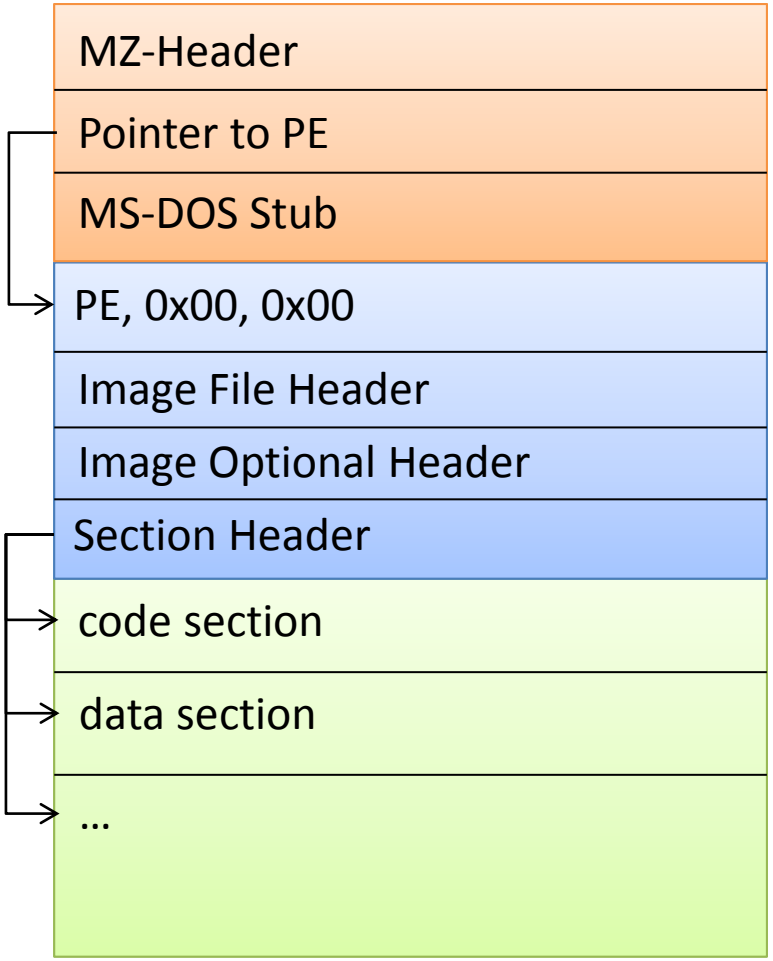


Image File Header

- Machine Type (ARM, PowerPC, AMD64, IA64, I386, …)
- Time/Date Stamp
- Amount of Sections
- Size of Optional Header
- Characteristics (Executable, DLL, 32-Bit Machine, Uniprocessor, …)

Image Optional Header

- Magic Value (PE, PE+, ROM)
- Size of Code, Size of Data, …
- Entry Point
- Image Base
- SizeOfImage
- SectionAlignment
- Checksum
- Amount of Data Directory Entries
- Data Directory

Data Directory Entries

- Export Table
  - Used e.g. by DLL's to export functions
- Import Table
  - Contains a list with DLL names and Function names
- Resource Table
- Relocation Table
- Debug

Section Header

- Name
- Size of Raw Data
- Address of Raw Data
- Virtual Size
- Virtual Address
- Flags

Example: „Hello World“

PE Header

- Image Base: 0x00400000
- Image Size: 0x4000
- Entry Point: 0x1000
- Section Header: .code, .data, .idata

.code section

1. Display a Message Box
2. Terminate

.data section

„hello“,0
„world,0

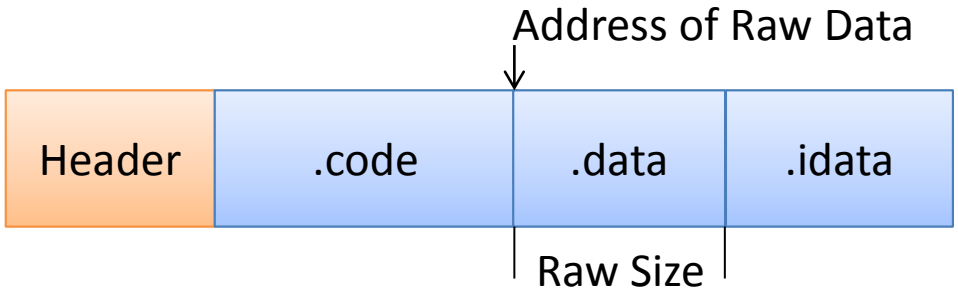
.idata section

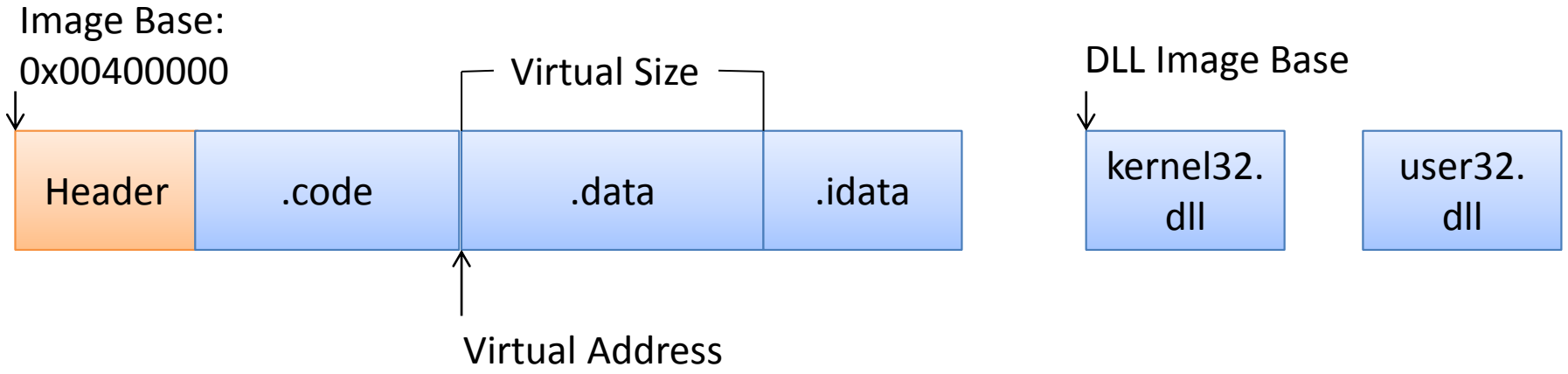
- Import ExitProcess from kernel32.dll
- Import MessageBoxA from user32.dll

PE Loading Mechanism

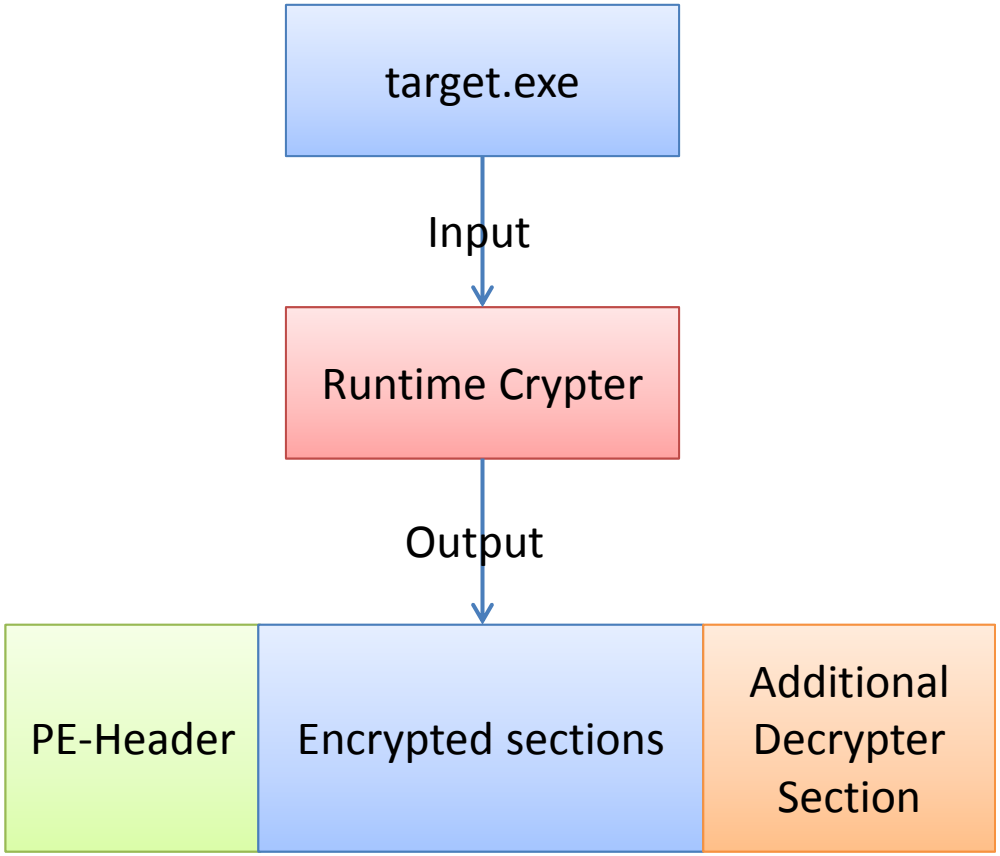
- Verify the MZ-Stub and the magic PE-Value
- Allocate memory at the image base address and copy the complete header into it
- Parse section headers and allocate memory for each section (size == virtual size) at the coresponding virtual address
- Copy the sections to their virtual addresses
- Parse Import Table and load the corresponding DLLs/APIs
- Set section Permissions and jump to the the entry point

PE-File on HD and in RAM

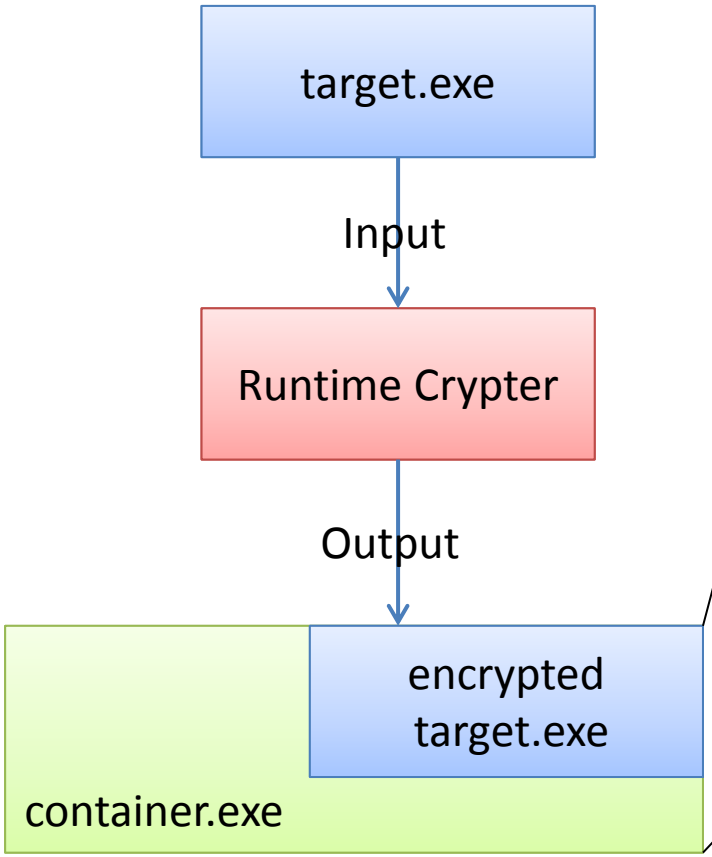




Possible Approach: Encrypt Sections

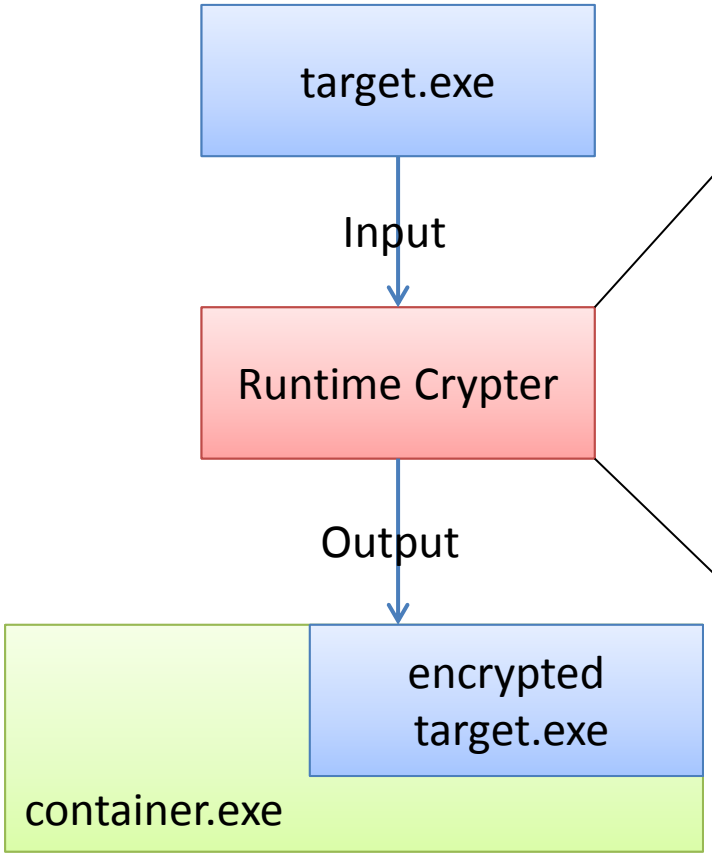


Possible Approach: Decrypt and Drop



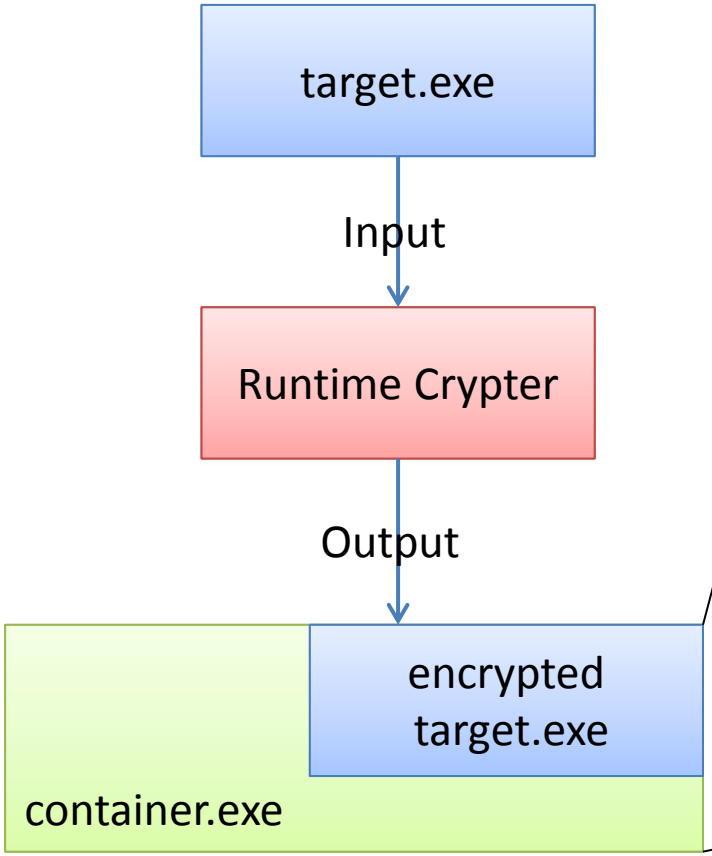
- Decrypt target.exe
- Drop target.exe on the HD
- Start target.exe with CreateProcess()

Our Approach



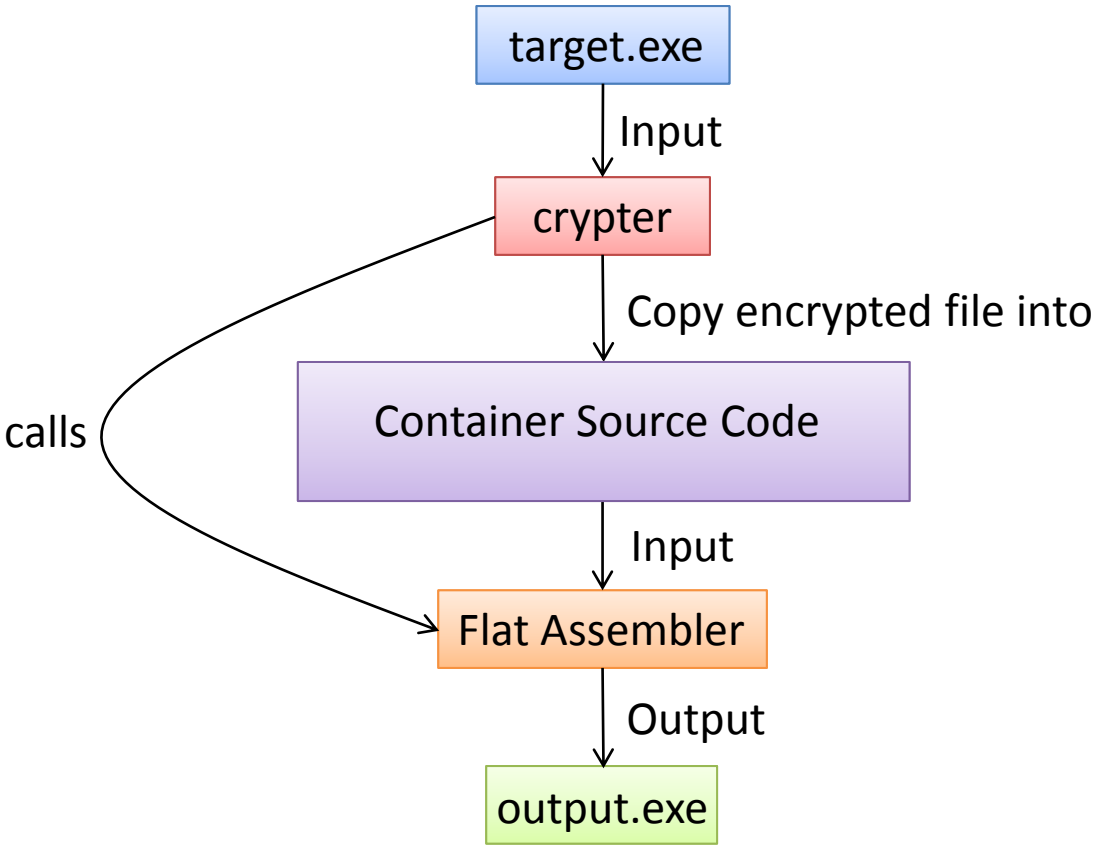
- Check whether input.exe is a valid PE file
- Encrypt input.exe
- Copy encrypted input.exe into container.exe

Our Approach

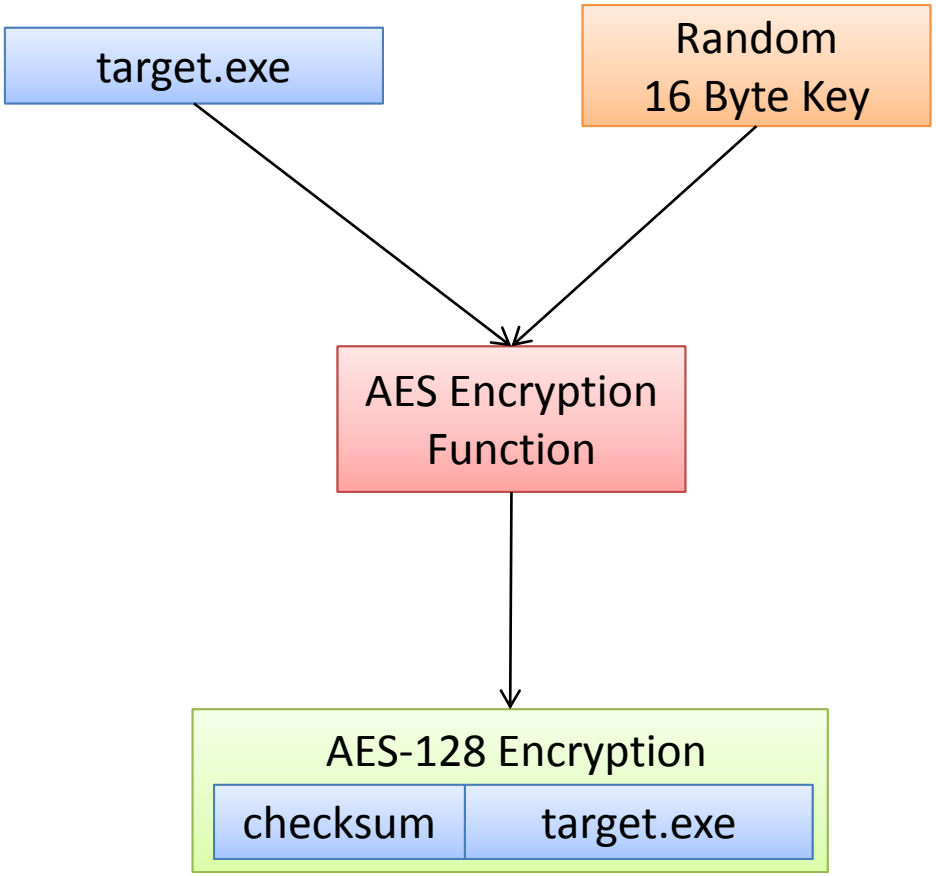


- Decrypt target.exe
- Load target.exe at its image base
- Load Sections and set section permissions
- Load DLLs and corresponding APIs

Generation of the container.exe



Encryption

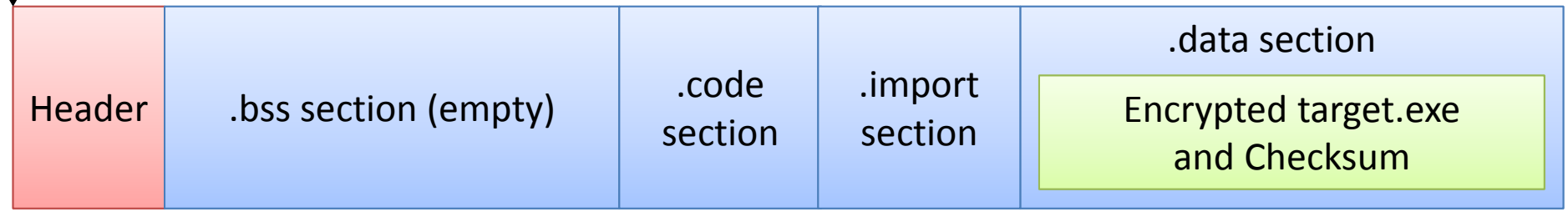

target.exe
Random
16 Byte Key
AES Encryption
Function
AES-128 Encryption
checksum
target.exe

Decryption: The file bruteforces itself

```
while(Keyspace.hasKey()){
        key = Keyspace.nextKey();
        decrypt(file, key);
        chk = checksum(file.target_exe);
        if(chk == file.checksum) break;
        else encrypt(file, key);
}
```

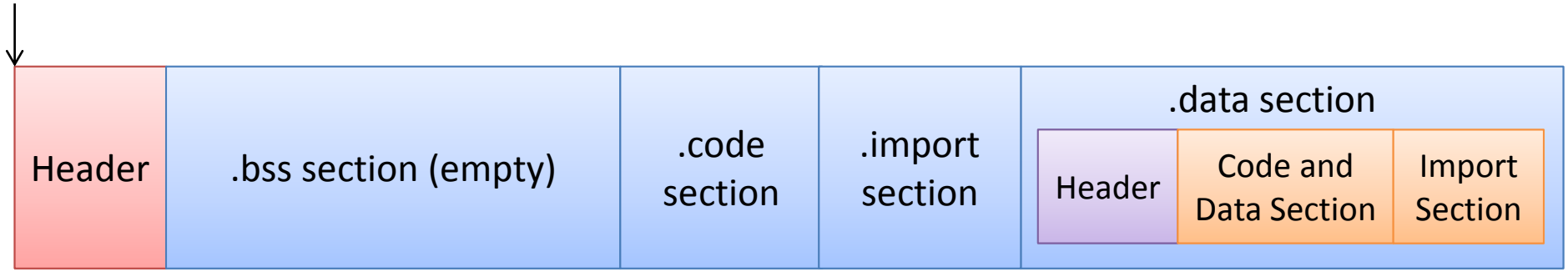
PE-Crypter Workflow

container.exe image base == target.exe image base



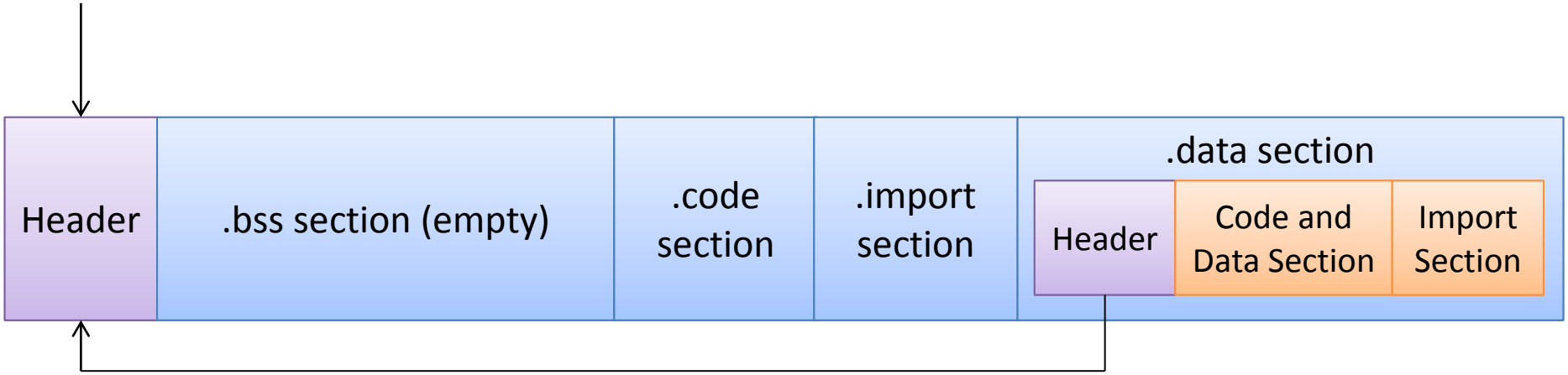
Decrypt the Target File

container.exe image base == target.exe image base

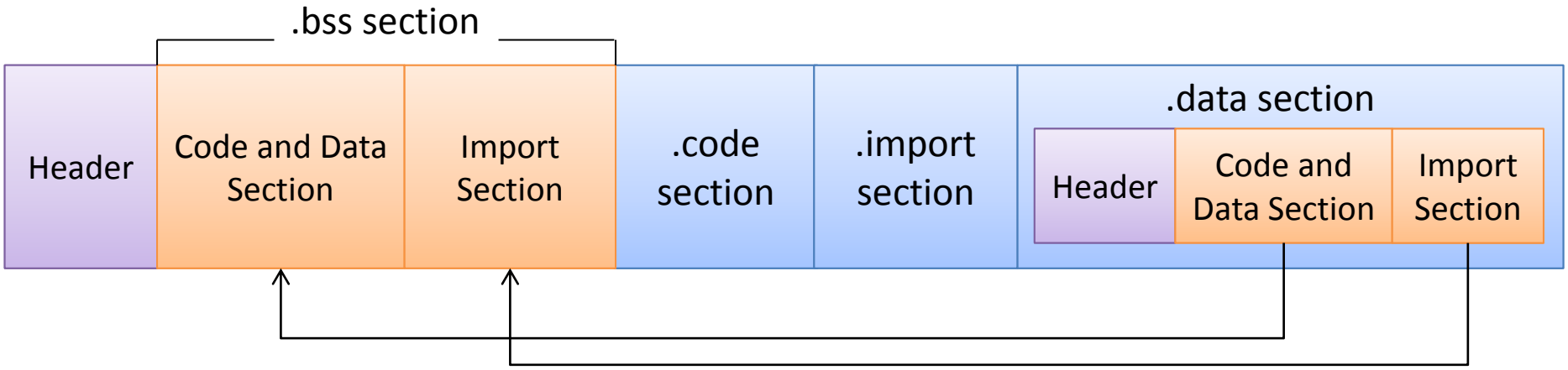


Load Target Header to its Image Base

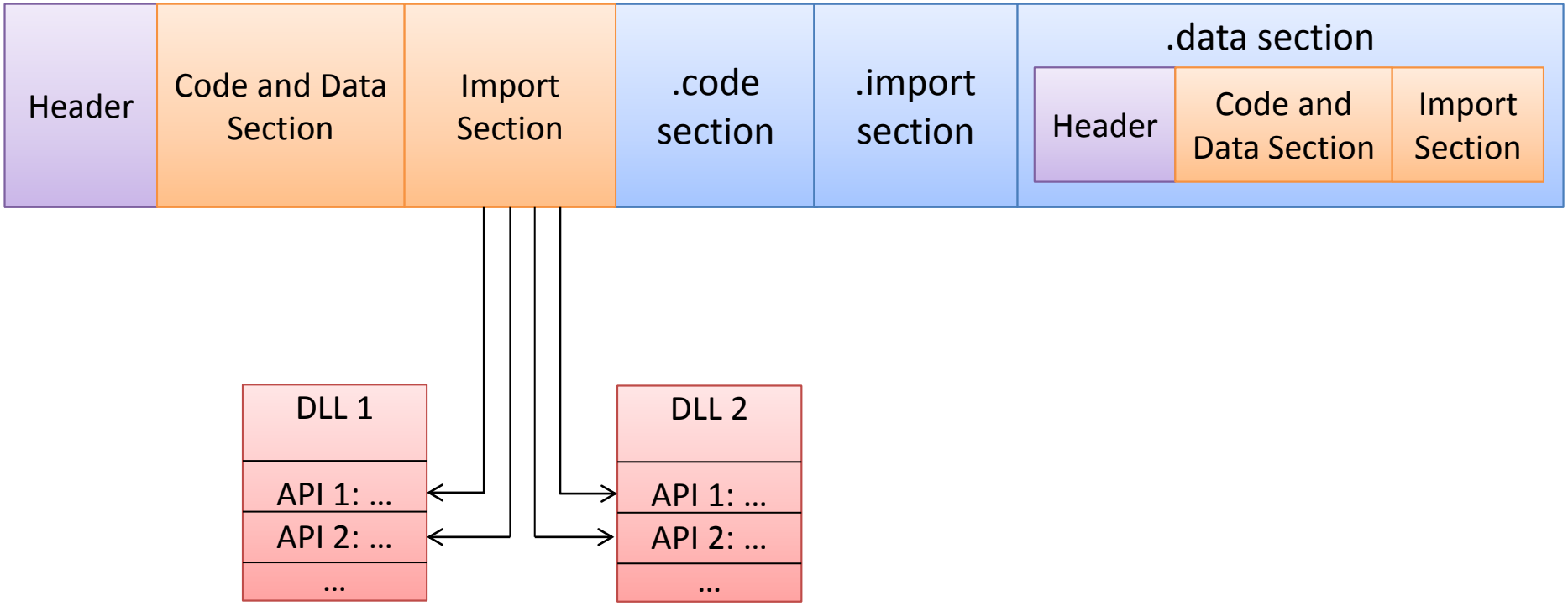
container.exe header is overwritten
with target.exe header



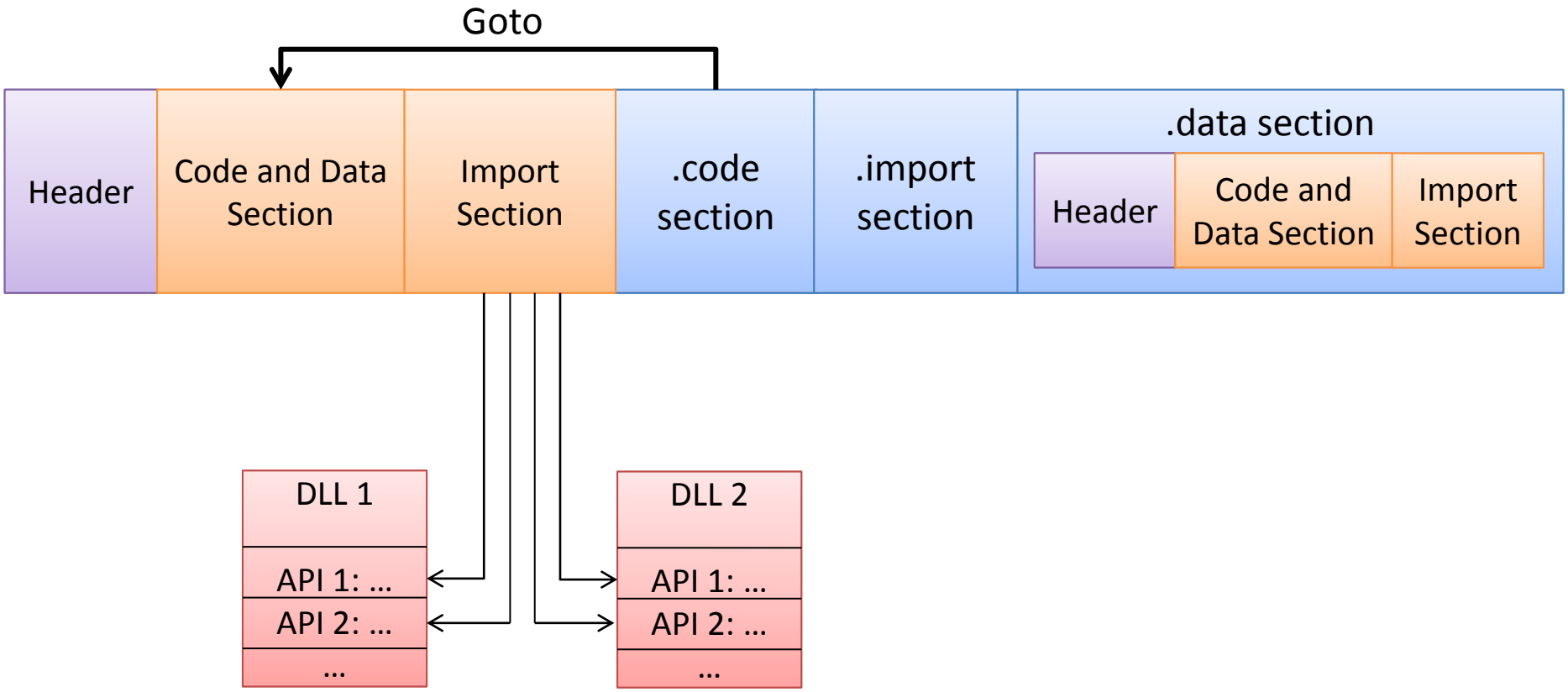
Load Sections



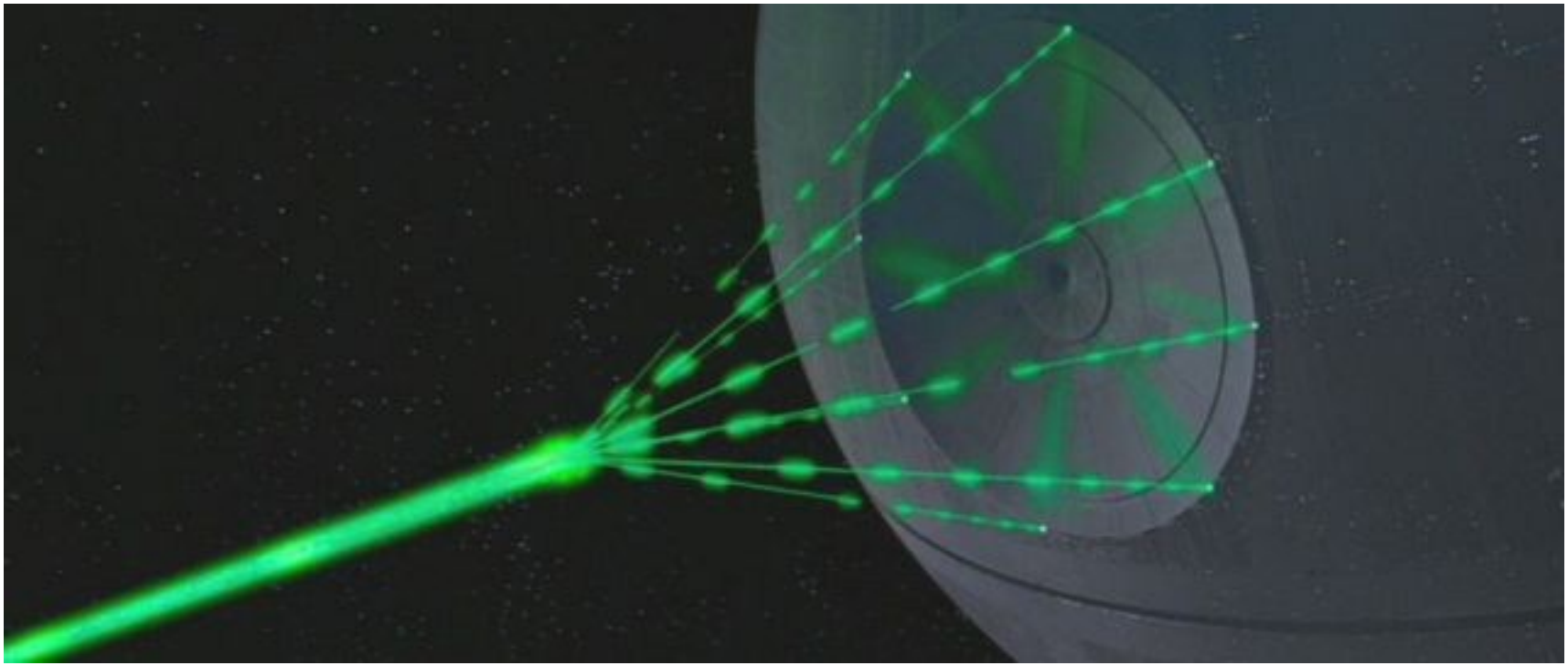
Load DLLs


Header
Code and Data Section
Import Section
.code section
.import section
.data section
Header
Code and Data Section
Import Section
DLL 1
API 1: …
API 2: …
…
DLL 2
API 1: …
API 2: …
…

Jump to original Entry Point



Demonstration



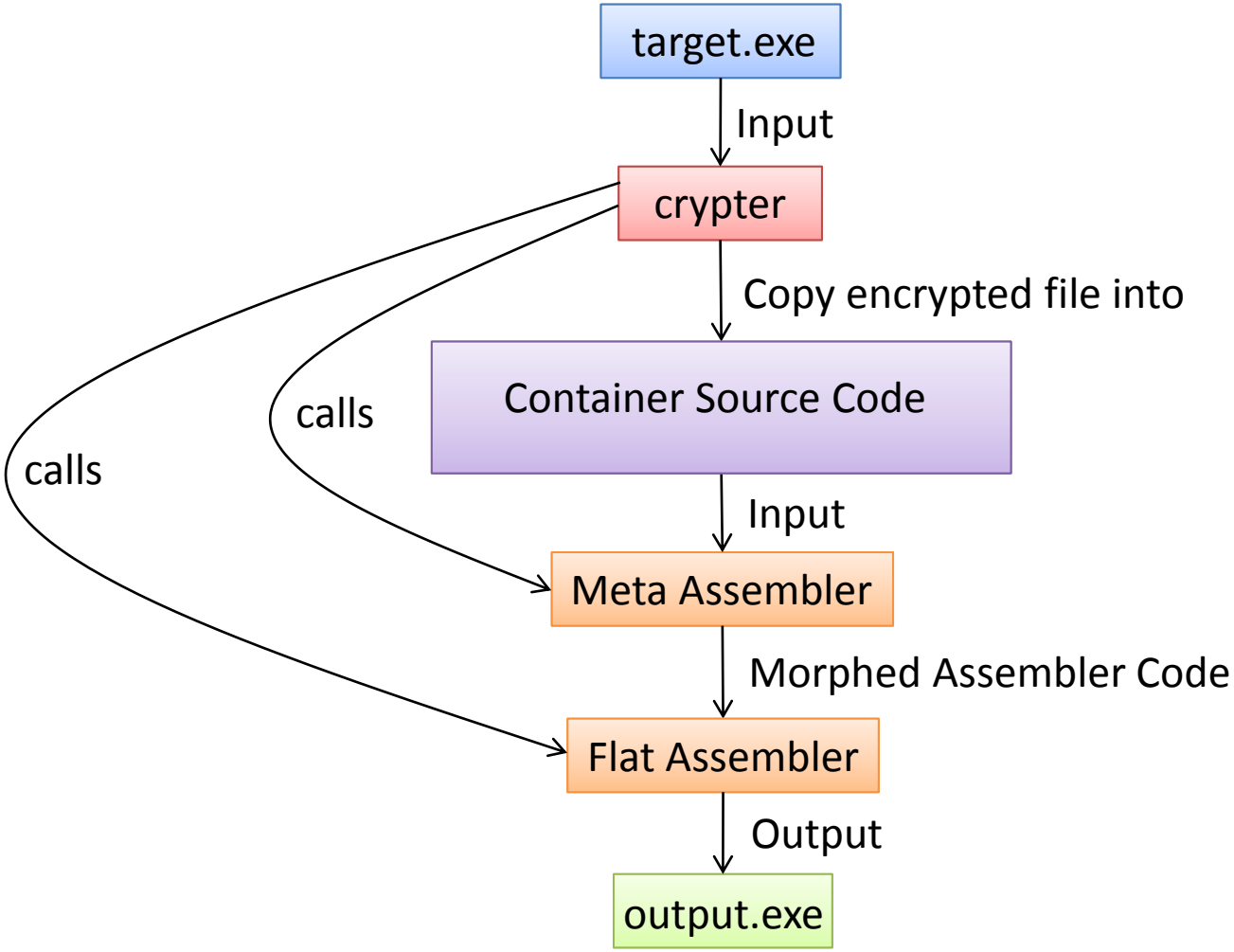
Further Work

- Anti-Heuristic (e.g. convert encrypted block to ASCII, …)
- Output.exe header morphing (e.g. fake API calls, …)
- Polymorphism or Metamorphism
- .NET support

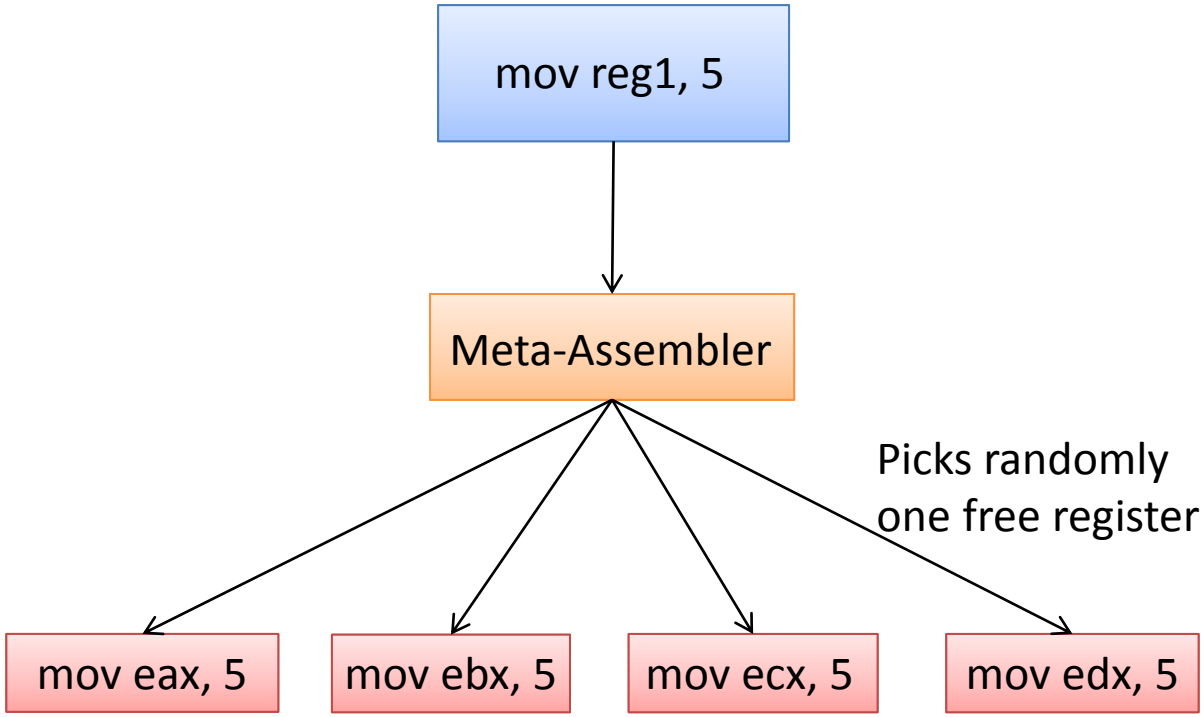
Meta-Assembler

- Assembler Language which is automatically transformed into Flat Assembler input language
- Inserts Garbage Code
- Random Registers
- Code Morphing

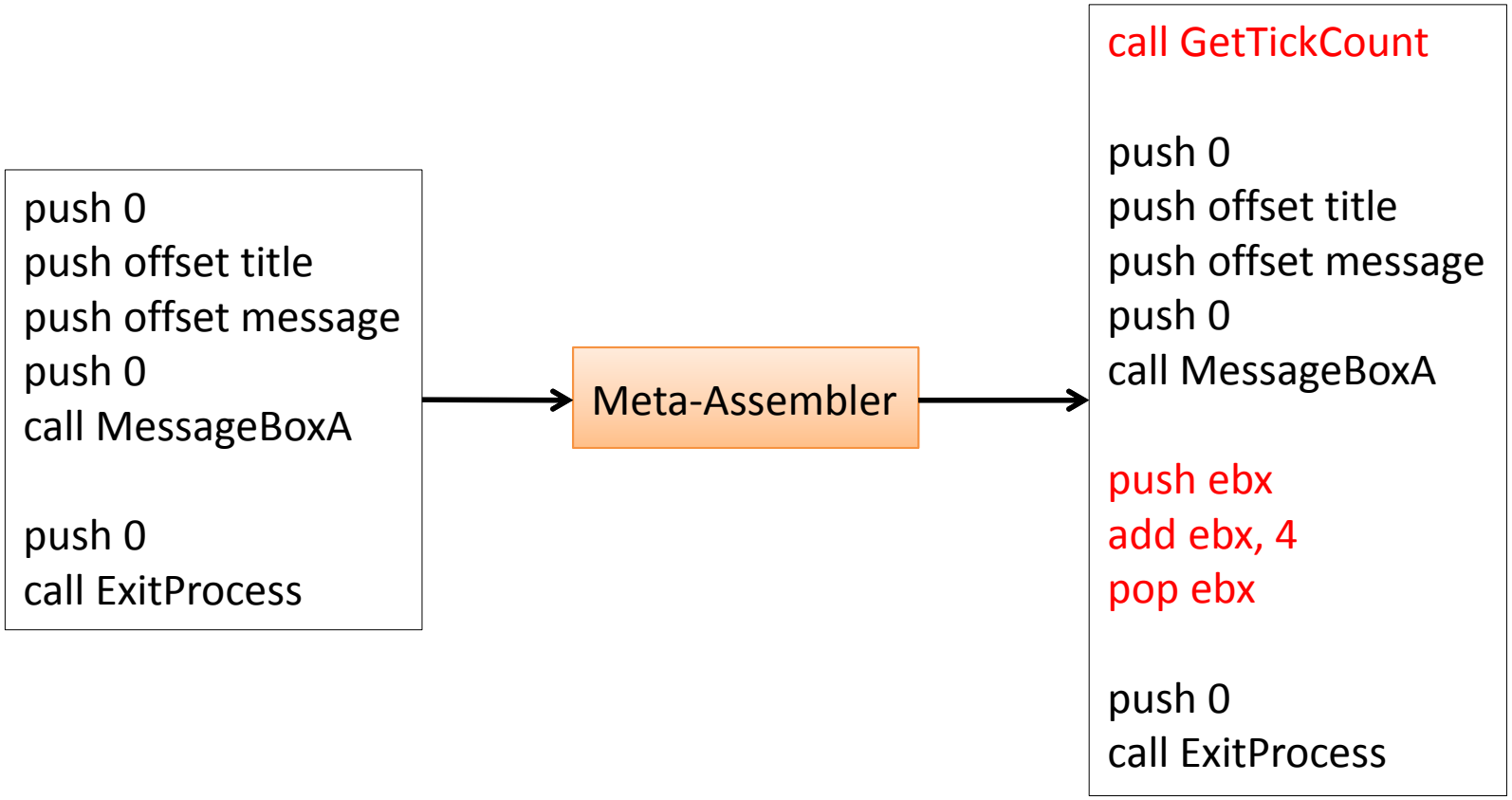
Meta-Assembler and the Crypter-Workflow



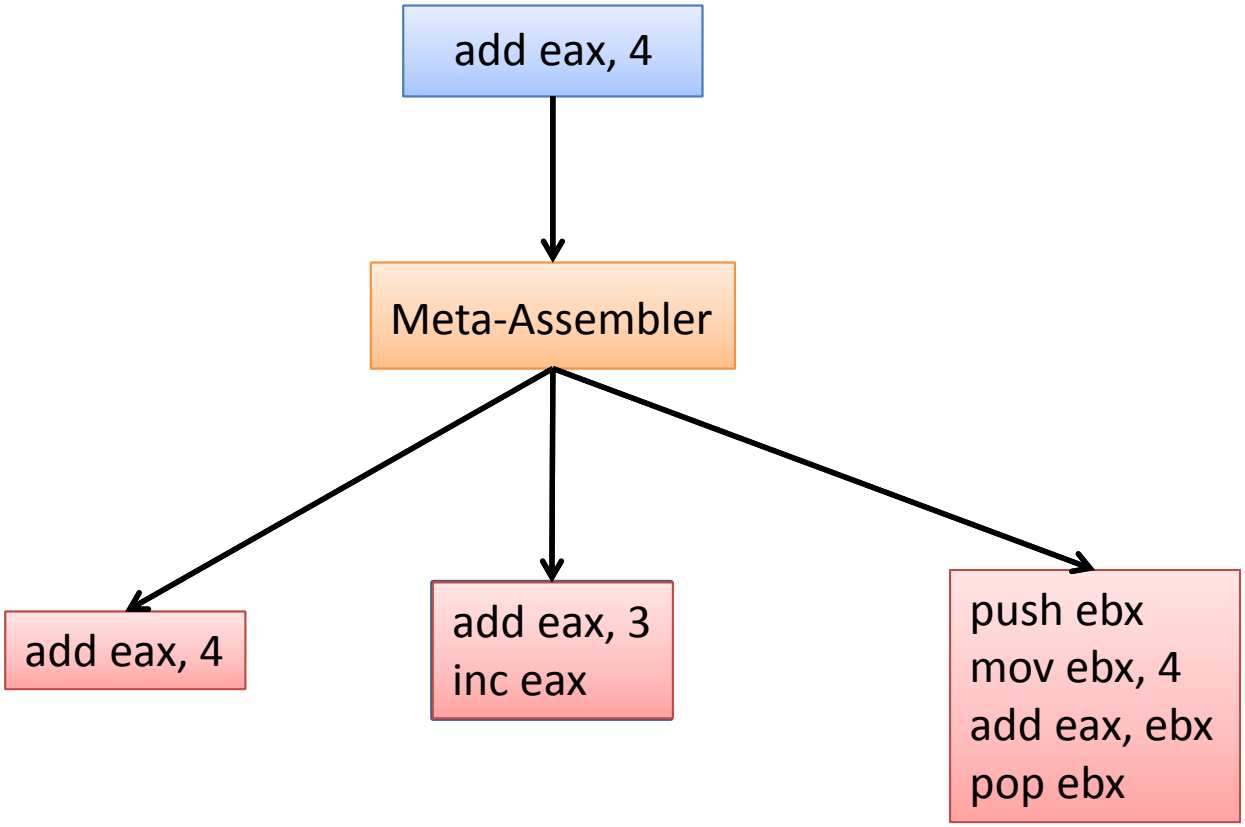
Meta-Assembler: Random Register



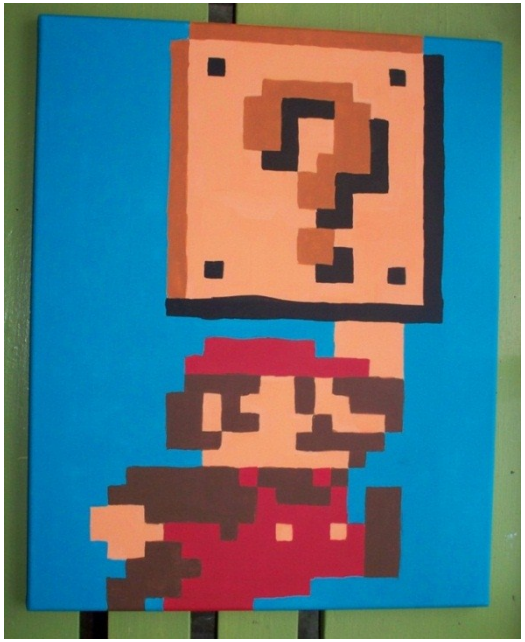
Meta-Assembler: Garbage Code



Meta-Assembler: Polymorpism



Questions



Mail: belial@nullsecurity.net

Slides: http://www.nullsecurity.net
Paper: http://www.nullsecurity.net
Crypter Source Code: http://www.nullsecurity.net