DeployMe

sleepunderflow

May 30, 2019

Abstract

This project is not to be used for malicious or illegal purposes. Project is available on a GitHub repository https://github.com/sleepunderflow/DeployMe

Table of contents

Chapter 1

Overview

The goal of this project is to create a tool that allows for easy and secure method of giving someone a bunch of tools and commands to run, collecting results from it (potentially in the future with some sort of scripting language allowing for things like conditional commands) and giving a single encrypted file for the person being helped to transfer to you for review

Disclaimer: This project is not intended to be used for malicious purposes and is only meant to make helping others easier, more efficient and protecting your knowledge.

Chapter 2

Injected Values

2.1 Definition

There is a structure in the client binary defined as follows:

```
struct sInjectedConfig {
  uint64_t header              = 0xDEADBEEFDEADBEEF;
  uint64_t flags               = 0x0000000000000000;
  uint64_t injectedDataOffset  = 0x1111111111111111;
};
```

It is statically initialised with those arbitrary values as it is meant to be overwriten by an injector. The header will be detected and then the corresponding values injected.

2.2 Fields

header - A magic value to be detected by an injector. It has to be at the beginning of the structure.

flags - A bitmap for enabling / disabling specific features. Explanation in section 2.3

injectedDataOffset - The offset in file to the beginning of the Main Header for the Embedded Tools (chapter 3)

2.3 Flags

Currently defined bit flags:

- **bit 0** - `FLAG_EMBEDPRESENT` - if set, Embedded Tools are appended to the file
- **bit 1** - `FLAG_EMBEDENCRYPT` - if set, the encryption of Embedded Tools is enabled. Otherwise just extract
- **bit 2** - `FLAG_ELEVATE` - if set, try to elevate privileges

The rest is currently reserved for a future use.

Chapter 3

Embedded Tools

This chapter explains the format for embedded executables for the client.

3.1 Format

Embedded executables and scripts are appended to the end of a client binary by the injector script.

The format of that section is as follows:

- Main Header
- Item 1 Header
- Item 1
- Item 2 Header
- Item 2
- ...

3.2 Main Header

The main header of embedded tools is defined as follows:

```
struct embeddedToolsMainHeader {
  uint32_t totalSize;
  uint32_t numberOfItems;
  char contentHash[64];
};
```

Explanation:

totalSize - the full size of embedded tools in bytes (including main header and individual headers)

numberOfItems - the number of embedded Items

contentHash - the SHA256 hash of the embedded part (Individual headers + items)

3.3 Individual headers

The individual header is separate for each embedded item and placed directly before the item content starts.

It is defied as follows:

```
struct individualHeader {
  uint32_t ID;
  uint32_t headerLength;
  uint32_t payloadLength;
  uint32_t flags;
  char fileName[64];
  char itemHash[64];
  char[] additionalData;
};
```

Explanation:

ID - numeric ID of the item. Starts from 0 and is being incremented by 1.

headerLength - size of the individualHeader structure including additional data in bytes

payloadLength - length of the item (excluding individual header)

flags - bitmap that contains settings per item. Currently used bits (bit order in file:|7-0||15-8| - 16-bit integer in little-endian format):

- 0: remove-after-use: 0-false, 1-true
- 1-31: reserved for future use

fileName - the file name of the item including extension

itemHash - SHA256 hash of the item after unpacking and decrypting

additionalData - extra metadata for the extractor in a form defined in section 3.5. The length of that data has to be `(headerLength-144)`

3.4 Items

The item itself is going to be encrypted using one of the individual keys generated and injected using the Injector script. It'll be decrypted when extracting. It'll only be extracted when will have to be used.

3.5 additionalData

Any additional metadata and information regarding the individual item is stored here as a continuous string, the format is (there's no spaces in between elements):

`opcode [length] data`

where opcode is a byte of the given value (in hex) and length parameter is only present for some opcodes.

Currently defined opcodes:

- **0x01 permission** - specify permission for the extracted item, only valid on Linux. Permission is a 3-digit number (ascii characters) representing permissions in chmod format (for example 644 or 777)

Chapter 4

Configuration File

The injector script is configured by a configuration file *config.conf* located in the same directory as the injector script itself.

Format for the file is as follows:

```
[Item]
configuration for item object

[Command]
command configuration

[Item]
...
```

There is no particular order for command/item entries. The only thing is that they will be added to the binary in the same order as entered to the config file but as they will be separated it doesn't matter whether you first add all the items then commands or command - item - command or however you like.

Configuration options are being added in format key="value"

if value should contain " character it has to be followed by \ (\")

4.1 Item

Item settings are in a following format:

```
[Item]
name="injected item name"
path="path to a file to inject"
permissions="permissions"
remove-after-use="true/false"
```

Path is a relative path to a file to be injected.

Permissions is a desired file permission in a chmod format (for example 755).

Name is the injected item name that can be used internally instead of the ID in the [Command] structure. Has to be a string and can't start with a digit.

If remove-after-use is set to true or missing, the extracted file will be deleted when the client terminates.

4.2 Command

Command settings are in the following format:

```
[Command]
command="command to be executed"
save-output="true/false"
print-output="true/false"
condition="condition"
loop="false/true"
```

Command is the desired shell command to execute. It can use the **{ID}** structure to use the injected items ([Item] part) If such structure is in the command, the required item will be extracted to a temporary location and {ID} will be replaced with the path to the extracted file. The ID is either an integer pointing to which [Item] in order is to be used starting from 0 or a string being the name of the required file ([Item]->name). If the name is used it will be replaced with the index while generating.

If print-output is false the output from the command will not be shown on the terminal. By default it's set to true.

If print-output is true (default) the report from running the command will be generated (chapter 6)

If condition is present, the command will only be run if the condition is true. The condition format is specified in chapter 5

If loop is true (implicit default is false) then the command will be repeated until condition becomes false

Chapter 5

Conditional Instructions

TO DO

Chapter 6

Report

TO DO

Chapter 7

Server

This chapter specifies how the server works and how to communicate with it.

7.1 Database

Server uses sqlite database in file `DeployMe.db`. This database has 4 tables:

- **users** - holds API keys that are allowed to create new items. API keys are randomly generated UUIDs. The key with ID==1 (one that is generated on running `createDb.py` is the only one authorized to create new users so be sure to save it)
- **items** - holds itemID, access API key and secret that are required to get the information from the server. Holds also the API key of the owner, and some of the encryption keys. (accessApiKey can only be used to get this particular item from the server, it does not work as user API key)
- **keys** - holds item encryption keys attached to the registered item
- **commands** - holds encrypted commands and flags assigned to the registered item

7.2 APIs

7.2.1 Response on error

Whenever the error happens in any of the functions the response is generated in the same format. Indication that the error happened is the value of the *"result"* field. If it's *"ok"* the query was executed successfully, if it's *"error"* the error occurred.

```
 {
"error":  errorCode,
"message":"Error message",
"result":"error"
}
```

- **error** - error code, should be the same as HTTP error code, most likely 500
- **message** - short explanation why the error happened
- **result** - always on error is "error"

All the formats below are only showing format on success

7.2.2 getencryptionkeys

Method: `GET`

Format: `/getencryptionkeys/<itemapikey>/<id>?secret=<secret>`

Response: JSON

```
 {
"commandKey":  "commands encryption key",
"itemID":  "item ID",
"itemKeys":  ["item key 1", "item key 2"],
"result":  "ok",
"returnKey":  "return encryption key"
}
```

- **commandKey** - encryption key with which commands stored in the database are encrypted
- **itemID** - item ID, for confirmation with request
- **itemKeys** - encryption keys for each individual injected item, as many keys as items, returned as an array
- **result** - on success always "ok"
- **returnKey** - encryption key with which client should encrypt the returned output so that it can be decrypted later (not yet implemented)

7.2.3 getcommands

Method: `GET`

Format: `/getcommands/<itemapikey>/<id>?secret=<secret>`

Response: JSON

```
 {
"commands":
[
{"command":  "command1", "flags":  1},
{"command":  "command2", "flags":  5}
],
"itemID":  "item ID",
"result":  "ok"
}
```

- **commands** - commands encrypted with commandKey from getencryptionkeys API call. Formatted as an array. Each command is encrypted with the same key. Each command is an object containing properties *command* which holds the actual command and *flags* which holds numerical flag for the command.
- **itemID** - item ID, for confirmation with request
- **result** - on success always "ok"

7.2.4 registernewobject

Method: `POST`

Format: `/registernewobject/<userapikey>`

Data: JSON

```
 {
"commands":
[
{"command":"command 1", "flags":0},
{"command":"command 2", "flags":1}
],
"numberofkeys":2
}
```

- **commands** - Plain text commands to be executed by client and encrypted already on the server. Each command is an object containing properties *command* which holds the actual command and *flags* which holds numerical flag for the command.
- **numberofkeys** - how many individual items are there, there will be a separate key generated for each

Response: JSON

```
 {
"accessApiKey":  "API key",
"itemID":  "item ID",
"itemKeys":  ["key1", "key2"],
"result":  "ok",
"returnKey":  "return encryption key",
"secret":  "secret"
}
```

- **accessApiKey** - item API key needed to get encryption keys and commands back
- **itemID** - new item ID, needed to get information back
- **itemKeys** - array of encryption keys, the number of them is the same as requested
- **result** - on success always "ok"
- **returnKey** - encryption key with which response from client will be encrypted (not yet implemented)
- **secret** - secret needed to get encryption keys and commands back

7.2.5 management/addnewuser

Method: `GET`

Format: `/management/addnewuser/<userapikey>`

Can only be called using first API key generated on DB creation!!!

Response: JSON

```
 {
"apiKey":  "user API key",
```

```
"newApiKey":  "new user API key",
"result":  "ok"
}
```

- **apiKey** - the API key used for user creation, just for validation
- **newApiKey** - newly generated user API key
- **result** - on success always "ok"

Chapter 8

FAQ

- **Will you add a function to run it quietly/fully automatically/send results automatically in the background?** No, this would make it much easier to use it for malicious purposes and that's not a goal of this project. This way at least a 2nd targeted party has to make some actions which makes for some form of a consent I guess.
- **What is the actual goal of this project?** No idea. Fun? Gaining knowledge?
- **Why aren't you using a library for that thing? Why don't you use X or Y instead of Z, it's easier?** One of the purposes of this project is to learn how things work under the hood. You won't learn using just ready-made tools so I'm trying to implement as much as possible myself.