Analysis of the 2015 ESET Crackme challenge

Lyxica
hello@lyxi.ca
@lyxi_ca

An introduction

In this write up I will be examining an ESET Crackme released in 2015. This crackme was released as a challenge to determine the problem-solving skills of reverse engineers. The goal of the challenge is to locate three different passwords, colloquially known as flags. These flags will be concealed in various locations in the challenge.

The tools I will be utilizing in this challenge are the following:

- IDA, short for Interactive Disassembler. This program is the state-of-the-art tool for reverse-engineering. The purpose of this tool is to translate native machine instructions into something that is human readable. I will also be using IDA as a debugger, which will allow me to run the program and suspend it, allowing me to examine the contents of the programs memory as I desire.
- dnSpy. This program will allow me to analyze programs written in the C# language. IDA could also be used for this purpose, however dnSpy is intended only for C# programs and provides useful tools that IDA doesn't.

Beginning from nothing

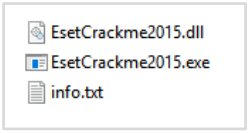


Figure 1. All files provided out of the box

In figure 1 I have listed all the files provided out of the box. The *info.txt* file contains an email address to submit a report to. Running the EsetCrackme2015 program, a simple interface consisting of three buttons and three text fields is shown (fig 2). Nothing happens when I click on any of the 'check' buttons, whether I input text into the fields or not. My first hint comes from looking at the currently running process list, which reveals that the ESETcrackme is starting a child process: svchost.

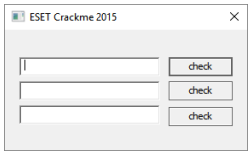
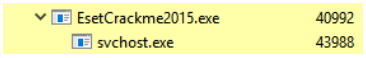


Figure 2. The ESET Crackme UI



Killing the svchost process causes the UI to disappear, however the ESET Crackme process remains active. This indicates that the UI code may reside inside svchost. Because svchost is a program used by the Windows OS to run services and the ESET Crackme has no legitimate reason to start a svchost process, it's likely that the ESET Crackme is injecting code into the svchost process. This is known as process injection and it's a common tool used in both legitimate and malicious programs. Legitimate programs will do this to enhance and add features to other programs. On the other hand, malicious programs use process injection to mask their presence on a system by hiding their code in benign processes.

Skimming through the EsetCrackme2015.exe and EsetCrackme2015.dll files in IDA, nothing of interest pops out. What I'm looking for at this point is any code that stands out. An example might be a function that uses the text "Invalid password". This stands out because this indicates that the function is involved, in some manner, with the processing of a password. However, the DLL file appears to be encrypted, and the exe file is full of functions that aren't very interesting. Before I will spend time analyzing those functions, I want to look for any other hints or interesting code that may exist. After skimming through the two files, there remains one program to examine: the svchost process.

When code is injected into a process, the injected code is typically inserted into a new segment of memory inside the process. For this injected code to take control of the process, the process must be started in a suspended state, its instruction pointer adjusted to point at the injected code, and then the process is resumed. It's necessary for the process to be started suspended because changing what code is being run after the process has started is extremely dangerous, and will almost certainly lead to the process crashing. Placing a breakpoint[1] on the Windows API responsible for resuming a process, I verify that the ESETcrackme is starting the svchost in a suspended state and then resuming it. This verifies my process injection theory, and I will have to extract the injected code to examine it in IDA.

Because the process would only be resumed after the injection is complete, I can easily extract the injected code using the breakpoint on the resume process API. Once the breakpoint is triggered, I can use IDA to examine where the processes instruction pointer has been adjusted to. Then, I can capture the segment of memory that the instruction pointer has been pointed at[2].

Finding Devin Castle

Performing the steps outlined prior, I'm able to save the injected code to a file. Opening this file in IDA reveals a program with no obfuscations of any kind. Examining the program's entry point, I find a dialogue-box function. This is a Window's OS provided function that causes a small window to popup. This would match the UI I see in figure 2. A dialogue-box function takes a handler function as a parameter, which is called every time an event occurs in the UI. The handler function's purpose is to handle events (such as button clicks) and to update the UI as needed in response to those events.

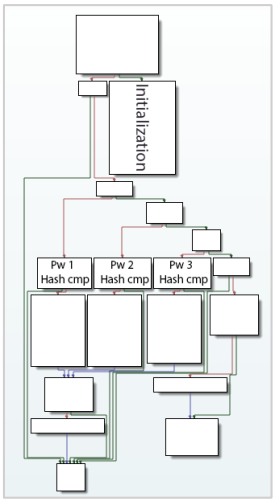


Figure 3. Graph overview of the event handler function

As the event handler function is responsible for reacting to events in the UI, it becomes a point of interest. Skimming through the code, I find it's responsible for checking the validity of all three passwords. Examining how it validates the passwords, I find three branches of code involved in checking input-passwords (fig 3). Each branch performs the same operation: it takes the input-password, hashes it using the SHA1 algorithm, and then compares it to a built-in value. Because the built-in values are SHA1 hashed, the comparison of the input-password does not provide much assistance in locating the original un-hashed values of the passwords. However, a hint may might be found by checking how the built-in hashes are set.

Searching for cross-references on the built-in hashes reveals they're set in a function located inside the initialization block (fig 4). Exploring this function, I find the built-in values are actually coming from the EsetCrackme program. When the UI first starts, it sends a request to the EsetCrackme process. The ESET process

[1] When an application is running, breakpoints can be set on code that cause the program to immediately stop processing when the processor tries to execute that code. A debugger attached to the process can then be used to examine the program's memory and state.

[2] Memory is assigned in chunks by the operating system. Attempting to access memory that hasn't been assigned to a program causes a fatal error. Because of this, I can easily look at a memory map, which is a list of assigned chunks of memory tracked by the OS. Because the injected code will be placed in a chunk that it itself requested, the entirety of that chunk will contain injected code.

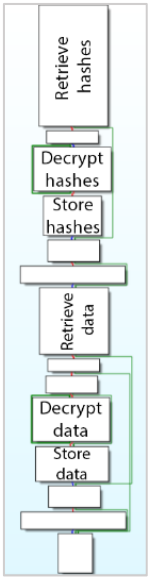


Figure 4. Graph overview of hash retrieval

then sends a payload consisting of three hashes that are encrypted. Next, the data is decrypted by the UI, and stored for validating the three different passwords. This unfortunately provides no assistance as the built-in password hashes aren't being calculated on the fly. However, I do notice something interesting. Beneath the code involved in request and decrypting the hashes, there are 3 blocks of code that do a similar task. Looking at them closely, I determine that they're involved in requesting and decrypting a string: "RFV1aV4fQ1FydFxk". This string is interesting, from appearance it looks as if it's either encrypted or hashed. If it was the later, it would have a distinct style compared to most widely used hashing algorithms[1].

I'd like to find out the purpose of this string, so I search for where the string is used. I can do that by placing a breakpoint on the memory storing that string. Anytime the processor tries to access that memory, the program will stop. After typing text into the first password field the program immediately stops. Interestingly, typing text into the other two fields doesn't cause the program to stop. After the breakpoint hits, I find myself inside a string comparison function. Examining the function that called the string comparison, I see that the string that "RFV1aV4fQ1FydFxk" is being compared this string: "RUNEV@=<". Looking for the source of this string, I find that it's a hashed version of my input in the first password field ("ESET").

Looking at the hashed value of "ESET", it appears that the hashing algorithm isn't very strong. A strong hashing algorithm would return a fixed length hash no matter how long or short the input text is. I'm curious as to how the hashed value will change as the input changes, so I decide to place a scripted code breakpoint on the string comparison function. This breakpoint will display, in a console, the hash of the text in the first password field. After recording the hash, the program will immediately resume. This will allow me to experiment with the text in the field and immediately see the hashed result.

After trying a few different strings (table 1), it becomes apparent that the hashing algorithm is quite weak. Another hallmark of a strong hash is that when even a single character in the input is changed, its output changes entirely. This algorithm doesn't exhibit this behavior, or the previous discussed fixed length behavior. Considering how changing a character only changes the closest characters nearest to it, I decide to brute force the hash. I try every combination of characters until the hash begins to match the target hash ("RFV1aV4fQ1FydFxk"). After a few moments of this, I'm able to find text that results in the target hash, "Devin Castle". Entering this text into the first password field results in a valid password, and with that I've found the first flag. Searching for similar types of weak hashes for the other two password fields does not yield any results.

Table 1. Inputs and their hashes

Input	Hash
ESET	RUNEV@=<
ESET Challenge	RUNEVBBCaFFrbFVtZ1U<
ESET_Challenge	RUNEVE9CaFFrbFVtZ1U<
ESET__Yellenge	RUNEVE9eWVVrbFVtZ1U<

After entering this password, a zip file is dropped in the same folder as the EsetCrackme2015.exe file. This zip file contains a driver, and a txt file which simply states "Install me." After installing the driver two additional files are placed in the same folder. They are "PuncherMachine.exe" and "PunchCardReader.exe".

[1] Example of the output of several common hashing algorithms. Thanks to https://www.pelock.com/products/hash-calculator

Punch Cards

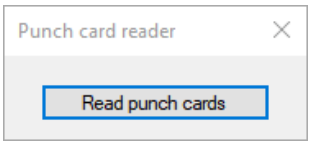


Figure 5. PunchCardReader's UI

Figure 6. PuncherMachine's UI

Now that I've found the first password, there are three avenues to explore: the driver, the puncher machine, and the punch card reader. Because reversing a driver requires a more involved process, I'll examine it last.

Opening PunchCardReader.exe shows the UI in figure 5. Clicking the "Read punch cards" button shows a pop up with a "Verification failed …" message. Opening the PuncherMachine.exe shows the UI in figure 6. Pressing the "Load punch card" button opens a file selection dialogue. There is no hint to the type of file it is expecting, and giving it a random file causes an error message that says "Invalid punch card!"

Opening the puncher machine in IDA reveals that it's a .NET program. Because of this I will use dnSpy instead of IDA. It's immediately apparent that: the control-flow has been obfuscated, all the names of the functions and classes have been renamed, and all the strings \in the program have been encrypted. Control-flow obfuscation is when you take code and break it apart into pieces that are arranged randomly. Code is then inserted at the beginning of the function which serves to direct what pieces of code are run and in what order.

The goal at this point is to find out what kind of file PuncherMachine is looking for, as well as see if there are any hints for the location of the file. Typically, I would search for the string "Invalid punch card!" to find what code is determining a valid file, but first I need to deal with the string encryption.

Figure 7 is an example of how a string has been replaced with a function called Bi() which returns a decrypted string. It accomplishes this by returning a specific subset of an array. This array is initialized from encrypted data sent by the ESETCrackMe program when the program first starts. After receiving the data, the program decrypts it and stores it in the array. At this point the array is already decrypted and the array contains 1,865 text characters.

```
stringBuilder.AppendFormat(<<EMPTY_NAME>>.Bi(), b);
```

Figure 7. A string has been replaced with a function that retrieves an encrypted string

126	BL()	Read timeout!
127	Bl()	Invalid argument
128	BM()	punchcard.bmp
129	Bm()	resource{0}
130	BN()	A.H
131	Bn()	Computer_card_punch

Table 2. Functions and their text outputs

Essentially all the strings in the program have been compressed into a single array, and when the program wants to access an original string it calls one of several (over 170) functions which extract specific subsets of the array.

Examining functions near Bi() reveals that there are 170 similar functions. This means that the array contains 170 strings. Obtaining the original 170 strings then involves copying the array as well as recording all the different subsets accessed by the 170 functions. The later is easily accomplished with a script that extracts the function names and the ranges used in said functions. This results in the data contained in table 2.

I get my first hint here, as I can see that there is a function called BM() that returns the text "punchcard.bmp." So, I'm looking for a bitmap image file called "punchcard.bmp".

I attempt a few different things at this point. I try using a white bitmap image made in paint. I look at the embedded resources on all the programs. I look for hints on where I could find this file in both the puncher machine and punch card reader programs. All of this leads to no new information, so I turn my eye to the driver and start examining what it does, and what makes it necessary.

Diving Through the Windows Kernel

The first place to start reading a driver in IDA is its entry point. This entry point registers functions in the driver to respond to different kinds of events. Additionally, the state of the driver is setup here. This typically involves registering physical or virtual devices, and setting up memory. Examining the entry point of the driver, I find a function that uses the following strings:

"EsetRam", "FAT1?", "ksidDmar", "SM-DRMAR", "IVE"

These strings are immediately interesting. They aren't any kind of typical strings you would find in a driver. And I can see that the second string looks like it's referencing the FAT filesystem. I'm curious how these strings are being used, so I place a breakpoint at the beginning of this function[1], and run the driver. When the breakpoint is hit, I find that the driver is assigning these strings to a chunk of memory almost 32 MB in size! This is very large for a memory chunk utilized by a driver. Additionally, I find that the "FAT1?" string gets placed in memory as "FAT16". This indicates that this 32MB sized chunk of memory may be a disk image. Examining the beginning of the chunk reveals a FAT16 header! I allow the driver to continue processing, and then mount the disk[2]. Inside the mounted disk I find missing image, however attempting to open the image results in an invalid image error. Attempting to open it in PuncherMachine also fails. Taking a closer look at the image in a hex editor reveals that the entire image appears to be random noise; another encrypted file.

The Driver's Read Handler

Returning to the driver, it may be that the image must be extracted a specific way. The first thing to check is the driver function responsible for handling file requests from the virtual disk.

When a program such as Windows Explorer requests a file from a disk, the operating system sends a request to the disk's driver asking it for the file. The driver fulfills this request and returns it to the operating system, which then returns the file to the program that requested it. Every driver, as part of its initial setup, must assign functions to handle requests like these. This makes it very easy to locate the code responsible for read requests.

Looking through the driver's function responsible for handling read requests, I find a very interesting trap. When the driver receives a request for the punchcard.bmp file, if a specific condition isn't met, it will instead return completely random data. However, when a specific condition *is* met the driver processes the punchcard.bmp file before returning it.

---

[1] Because the driver will be placed at a random address in memory, the first thing I do is replace the first byte of the entry point with the byte 0xCC. This will cause the CPU to get trapped once the entry point code starts running. Once I have control in a debugger, I can rebase the IDA image to reflect where the driver is located in memory.

[2] I did this by utilizing a directory symlink. I symlink'd the devices UNC path (\\?\GLOBALROOT\Device\45736574\) to a folder, which allowed me to access the disk in windows explorer.

The condition that this trap relies on is the existence of a key in the registry. When the driver does its initial setup, it creates a background thread that continuously monitors for the existence of a key called "ESETConst". If the driver is able to find and read this key, then the condition passes. However, if the key does not exist, or the key does not contain any data, the condition fails and the driver returns random data.

With this information, I create the missing registry key and fill it with the text "ESET". This causes the condition to pass, and the driver begins to process the punchcard.bmp file in 512-byte sized chunks. As I watch the driver begin to process the file, I discover something very interesting: the driver is not processing the image at all! The 512-byte chunks are passed to code that is outside of the driver. Investigating the source of this code, I discover that as part of the driver's initial setup, a request is sent to the ESETCrackme2015 program. This request causes the ESET Crackme to respond with 8kb of code, which the driver stores for future use. This code is then run to process the punchcard.bmp file. This is something that virtually no normal program does, and its only real purpose is to hide said code. Because the code isn't located in the driver file, you wouldn't know about it until you stumbled upon it. I'll save this hidden code to a file and begin analyzing it in IDA.

Analyzing the hidden code

Now that I've extracted the code the driver uses to process the punchcard.bmp file, I can analyze it using IDA. For simplicity, I will refer to this code from now on as its own program. The reason is because the code itself doesn't depend driver, or the ESETCrackme2015.exe program. For all intents and purposes, the code is its own little program doing its own thing.

Skimming through the program in IDA doesn't reveal very much. The program contains 67 different functions, but they're all cryptic, and provide very little context for what they are accomplishing. None of the functions utilize any text, nor do any of them use any of the window's API. Without any hints as to the purposes of the different functions in the program, analysis requires much more time.

My first step when trying to figure out a cryptic program is to run it multiple times in a debugger. While I'm debugging, I'll step through the programs code as fast as I can. The purpose of this is to get a feel for what the different components of the program are accomplishing. Doing this also allows me to catch interesting memory addresses the program may be accessing. For example, some programs hide the text they utilize by decrypting text only at the moment they need to use it. At that point the string would be sitting in memory in an decrypted state and can be noticed by a careful eye.

After running through the program, I am able to determine roughly three stages.

Stage 1

The program reserves and clears 1 KB of memory. This memory is then used in almost every single function inside the program. Here is some of the things stored in this memory:

1. A list of functions inside the program. The list's size is 256 elements, but only the first 16 items in the list point at unique functions. The rest of the items in the list point at the same function.
2. The location in memory of the punchcard.bmp image
3. A flag that indicates when the dispatch loop in stage 2 should stop processing.

Stage 2

Once the memory has been setup, a "dispatch loop" is run that uses the function list from stage 1. The dispatch loop determines which function from that list to run, runs it, and loops. This loop does not end until one of the functions it calls tells it to stop. Out of curiosity I recorded how many iterations the loop goes through, and I recorded 35,000 iterations before I stopped recording.

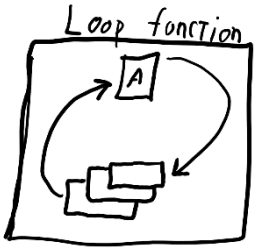


Figure 8. The block labeled A determines which function to run.

Stage 3

The program cleans up any memory it requested from the operating system and returns back to the driver, where the processed punchcard.bmp file is returned.

After establishing this information about the program, I run into a wall. The thing is that very little actually happens outside of stage 2. In fact, almost all of the time the program is running is spent inside the stage 2 dispatch loop 2. This presents a significant analysis issue, as the functions called by the loop are basic. For example, one of the functions called by the dispatch loop simply adds two numbers together. The reason this is an issue is that the *real* code that I need to analyze has been effectively hidden away inside dozens of functions. This is another form of control-flow obfuscation, except this form of obfuscation is harsher. Instead of a program executing 500 lines of code, it executes 500,000 lines of code.

After examining how the dispatch loop determines what function to call next, I notice that a bundle of data is passed from the driver to the program. The dispatch loop uses this data by reading from it a byte sized number. This number is then used to determine which function in the list from stage 1 should be called next. Running through the dispatch loop, I discover an extremely important detail: not only does the dispatch loop access this data, but almost every function called by the dispatch loop does too! It's at this point that I realize the extent of the obfuscation: it has been obfuscated with a virtual machine.

Virtual Machine Obfuscations Explained

Most programs are written in a programming language that requires the source code to be compiled. This compilation process converts the programming language into instructions that a specific architecture of processor can understand. These instructions, also known as opcodes, perform very specific operations in a processor. For example, one of the most common instructions for intel-based processors is the "mov"

instruction[1]. This instruction **copies** data from memory to register[2], from register to register, or from register to memory.

When a virtual machine-based obfuscation is applied to a program, these native instructions are replaced with **virtual instructions**. These virtual instructions cannot be read by the processor and therefore mean nothing to it. Instead, these virtual instructions are fed into a **virtual processor**. The virtual processors job is to translate the virtual instructions into instructions that the real, physical processor can understand.

Imagine a book written in English, translated into another language only understood by a single person. That one person could recite and translate the book back into English in real time, but no one else could. This is where the massive difficulty comes from: trying to decipher a language spoken by only one program. In this example, IDA is only able to understand programs written in English. If I want to decipher the program being fed into the virtual machine, I will have to teach IDA how to understand the virtual machines language.

Deciphering the Virtual Machine

The program I was talking about previously is actually a Virtual Machine. In order for the virtual machine to accomplish anything, it must be fed a VM program. The goal from this point is to understand what the VM program that is being fed into the virtual machine is accomplishing. After all, I don't really care about the virtual machine itself.

The first step is to save the VM program to a file. Finding the VM program is easy, as that is what is being accessed in both the loop dispatcher, and the functions the loop dispatcher calls. Saving the file, I label it as "VM 2B5". This is because the file's size is 0x2B5 (693 in decimal).

My process for reversing the virtual machine instructions is to place a breakpoint on the dispatch loop before it calls a function. Using the dispatch loop as a guide, I will first find out where the instructions start in the VM 2B5 file. This allows me to find a 12-byte header at the start of the file[3]. Next, I will find out what the first instruction in the VM program accomplishes. Following the dispatch loop into the first function called, I need to determine what the *operands*[4] to this instruction are. During this process I discover that the operands can vary in size, from one byte to four bytes. It's critical to parse the correct sizes for the operands, because if IDA reads too few or too many bytes, it will mess up IDA's ability to properly understand the program[5]. Once I've recorded the size of the operands used, I look at the function itself. It seems to be performing a "CMP" operation. A "CMP" instruction compares two values, and returns a number indicating which value is largest. If both values are the same size the "CMP" instruction returns the value 0. With this I'm able to start writing my plugin for IDA. My strategy here is to add each opcode to IDA, one by one. This will allow me to see how the file changes as I add more opcodes. In particular, I'm doing this because I need to know if I'm recording

[1] https://www.strchr.com/x86_machine_code_statistics
[2] A register is a small data storage space located inside the processor. Registers typically can contain four to eight bytes of data. Registers are used because they have access times measuring a nanosecond, whereas reading data from memory can take at least 50 nanoseconds.
[3] The first instruction read in the dispatch loop is located 12 bytes after the start of the VM 2B5 file.
[4] An operand is data that the instruction uses to do its job. For example, in "ADD 5, 3", the addition would be the operation and the 5 and 3 would be the operands.
[5] Fun fact, determining the size and type of operands is one of more complex components inside a processor.

operands correctly. If an instruction uses operands in a way I misunderstood, then instructions that IDA could read before will instead start appearing as invalid. This will be critical in preventing bugs. This process of adding instructions and testing them continues until I've added 17 instructions. At this point IDA is able to read the VM 2B5 file without any issues, and I get my first glimpse at VM 2B5's code.

Finding Barbakan Krakowski

At this point, I know the following things:

- A virtual machine is setup by the driver. This virtual machine is invoked only when the driver attempts to read the punchcard.bmp file.
- To process the punchcard.bmp file, the virtual machine runs a program that I have dubbed VM 2B5. This program processes the punchcard.bmp file in 512-byte sized chunks.
- My plugin for IDA is able to parse the VM 2B5 code without any issues.

With all that out of the way, I finally have a view into what the VM 2B5 file looks like (fig 9). Because the graph is nicely laid out with clear indications of looping structures, I'm very confident that my IDA plugin is correctly interpreting all of the programs code.

Running through this program, I observe the following things:

- The string from the registry key (mentioned at the end of "The Driver's Read Handler" section) is used.
- Only the right side of the program runs.
  - In figure 9, the first block of code that is run is located at the top-right. Immediately after, it splits into a left and right side, with both ending at the bottom of the graph. Everytime I've run the program I have only ever seen it run code on the right side.
- There is an 18-byte array contained in the program.
- There is a small string ("ESTE") used inside the program.

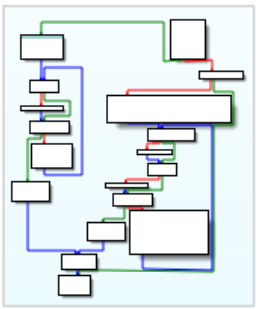


Figure 9. Graph overview of VM 2B5

Running through the program, it appears that it modifies the 18-byte array using data from the registry key string. To do this the following steps are taken:

1. Loop simultaneously through the 18-byte array, and the string from the registry key.
   a. In each loop iteration take one byte from both arrays and XOR them together, then add the number 1 to the result.
   b. Take a character from the "ESTE" string, and then XOR it with the result from above.
   c. Finally, bit-rotate the result from above once, to the left. Replace the byte that was read from the 18-byte array with the bit-rotated value.

After the 18-byte array is processed it is passed to a different VM program. Before venturing into that program, there is one unanswered question: what does the left side do? As I noted at the start of the section, only the code on the right side of the graph (fig 9) actually executes. In these kinds of challenges flags could be hidden anywhere, including in branches of code that are otherwise not normally accessible. Because of

this, the left side peaks my curiosity. After manipulating the VM to force it to run the left side of the graph, I find it performs almost the same actions as the right side, however it doesn't use the registry string, and it uses a different array (also 18 bytes). After allowing the code to run I find that the alternative array has been modified into a string that reads "Barbakan Krakowski". Entering this into the 2nd password field returns a successful result. Only the final password remains!

Decrypting the punch card image

As I mentioned above, once VM 2B5 finishes processing the 18-byte array, it passes it to a new VM program. This program is actually contained inside VM 2B5, but in an encrypted state. Its 0x137 (311 in decimal) bytes in size, so I'll call it VM 137. To run this program, a new VM instance is created. Next, the new VM instance runs a small section of code contained inside VM 137 that decrypts a large segment of code. Finally, the VM instance runs this decrypted segment of code and passes it the 18-byte array from above.

Going through the code in VM 137, it doesn't take very long to realize that VM 137 is an implementation of the RC4[1] cipher. There is nothing special about this implementation of the RC4 cipher, and there are better resources for understanding what RC4 does, so I won't explain it beyond stating that RC4 is an algorithm used to encrypt and decrypt data.

With this information, the structure of the VM programs makes sense:

- VM 2B5 contains a program that creates a cipher key.
  - The 18-byte array I mentioned in the previous chapter is the decryption key used to decrypt the image. As it exists in its normal state, the decryption key is itself encrypted. Before the key can be passed to the RC4 cipher, the key must first be decrypted. The string from the ESETConst registry key is used to decrypt the RC4 key.
- VM 137 contains an implementation of the RC4 cipher.

One thing remains: because the decryption key must be itself decrypted, I must find the correct text that should be in the ESETConst registry key. If I use the wrong text, the rc4 key won't be decrypted, and the RC4 algorithm will return garbage data. The next step then is to find the correct text that will properly decrypt the decryption key.

Attempting to discover the decryption key, I try the following:

- Using "Barkakan Krakowski", "Devil Castle", "ESET", "ESETConst", "ESETCrackMe2015".
- Searching for more VM programs. My thought is that the key may be hidden in a different VM. I was able to locate additional VM programs, which were residing in the ESETCrackMe2015.exe program, however they were not relevant.
- Looking for hints in the PuncherMachine and PunchCardReader programs.
- At this point in my analysis, I am aware that ESETCrackMe2015.exe is responsible for dropping the PuncherMachine, the PunchCardReader, and the driver files. I also know that when the driver runs, it requests the code for both the virtual machine and its programs from the ESET program. It occurs to me that the decryption key may reside in the ESET Crackme as a file that doesn't normally get accessed, similar to the normally inaccessible branch from VM 2B5. After some time, I determine how the ESET

[1] http://www.crypto-it.net/eng/symmetric/rc4.html?tab=0

Crackme handles requests for data, and locate 21 resources that have been embedded into the ESETCrackMe201.dll file. Unfortunately, none of the files contain any relevant information.

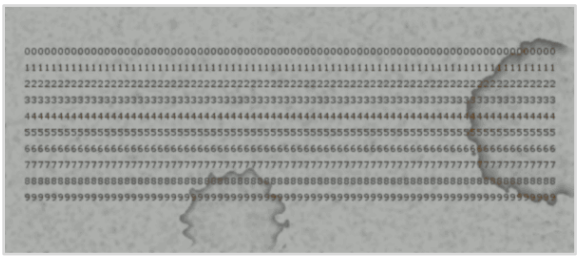
One idea I had but didn't attempt, is to brute force the key. This is typically not possible due to how long it would take, however there is a major weakness in the implementation of the encrypt. The weakness is due to how the driver calls the VM. Remember how in "The Driver's Read Handler" I described how the punchcard.bmp file is processed in 512-byte sized chunks? This means that the punchcard.bmp file, a file that measures 77,866 bytes, must be processed 152 times. Here arises the problem: *each 512-byte block is decrypted using the exact same key.* This means that if I am able to determine the correct byte that decrypts the 1st byte of a 512-byte sized block, I can use the same exact byte to decrypt the 1st byte of *every* block. This is a massive flaw in the implementation of the encryption, and this flaw is compounded by the fact that bitmap files have a very well-defined header. With some effort I could determine the first 50 bytes of the block! I decide not to attempt this approach, the reason being that I'm pretty sure this is not the intended way, and while bypassing the need to find the key is a valid strategy, it's a strategy that doesn't solve the mystery.

On a whim, I become curious if there might be more to the "Barbakan Krakowski" string then I thought. I decide to create a script that brute forces the 18-byte array from VM 2B5. What it does is it implements the algorithm detailed in VM 2B5, however instead of taking characters from the ESETConst registry-key, the script will instead repeat the algorithm for every ASCII character. Each time the algorithm finishes, the script will check if the result matches a character in "Barbakan Krakowski". If it does, the script saves the original character that caused that result, and then moves onto the next position in the array.

Here's an example of it in action:

Using the first byte of the embedded array, run the algorithm and pass it the character "A". If the result if the algorithm is the character "B", store the character "A". If the result is not "B", run the algorithm again but with character "B", then "C", "D", and so on until either every ASCII character has been tried, or a matching result has been found. Once either a result has been found, or every character possible has been tried and nothing found, move onto the second byte of the array and start the search again. This time the script will search for a character that causes the algorithm to output the character "a".

After running the script, I get a lovely message that reads "Reversing is great". Using that as the decryption key, the RC4 algorithm is able to successfully decrypt the punchcard.bmp file, revealing this wonderful image.



The Puncher Machine

Figure 10. Puncher Machine UI after being given the punch card image

For a quick recap, at this point: I've found the first and second password; I've decrypted all the strings in both the puncher machine and punch card reader programs; and I know that both programs have been obfuscated. The most important obfuscation used is code-flow obfuscation.

After opening the puncher machine and giving it the decrypted photo, I see the UI in figure 10. From this screen I must input two calibration codes for the program to continue. Entering random text causes an error message that displays "Calibration error!" After searching for this error message, I locate the function that reads both calibration codes and returns a true or false value based on them (fig 11).

Because the function reads both calibration codes, and returns a truth value based on those codes, it's likely that the puzzle will be found inside this function. I can test this assumption by modifying the function to always return true, and then observing how the program changes. Running the program with this modification, entering any text causes a new UI to show, proving my theory correct. Unfortunately, the shortcut doesn't work, as interacting with the new UI causes the program to crash. I won't be able to just skip the calibration code puzzle.

My first step will be to de-obfuscate the function's code. This involves removing the code-flow and name mangling obfuscations I described in a previous section. Doing this by hand doesn't take too long, and soon I get the result in figure 11.

```
public bool CalibrationCodeIsValid(string calibration_code_1, string calibration_code_2)
{
    if (calibration_code_2.Length > ASCII_Array.Count) { return false; }
    try
    {
        // This block uses calibration code 1
        List<uint> list = new List<uint>(); // OPCODES
        for (int i = 0; i < calibration_code_1.Length; i += 8)
        {
            list.Add(Convert.ToUInt32(calibration_code_1.Substring(i, 8), 16));
        }
        UnknownDelegate dynamicFunction = Unknown_Function(list.ToArray());

        // This block uses calibration code 2
        for (int i = 0; i < ASCII_Array.Count; i++)
        {
            string input;
            if (i < calibration_code_2.Length)
            {
                input = ASCII_Array[i].ToString() + calibration_code_2[i].ToString();
            } else
            {
                input = ASCII_Array[i].ToString();
            }
            var result = dynamicFunction(input);
            if (Unknown_table.ContainsKey(result))
            {
                Table_generated_from_calibration_codes.Add(ASCII_Array[i], Unknown_table[result]);
            }
        }
        return true;
    } catch (Exception) { return false; }
}
```

Figure 11. The de-obfuscated function that reads both calibration codes

First off, there are two conditions that cause the function to return false/fail. First, if calibration code 2 is too long the function will fail. Second, if an exception is thrown inside the function, the function will fail.

Starting with calibration code 1, I can see that the calibration code gets split into eight-character strings. Next, the strings are parsed as numbers and stored in an array. After processing calibration code 1, the array is passed to the function *Unknown_Function*.

Looking through *Unknown_Function* to determine how the array is used, I find an interesting development. The array gets passed

to a function called ***createMethod***, however this function doesn't reside inside the PuncherMachine program. Instead, this function exists inside a DLL file that has been stealthily loaded into the PuncherMachine program. Examining the DLL file, I find that it contains only two functions: ***createMethod*** and *create_HashMethdForOpcodes*. When I look to see what *createMethod* does, I find something very peculiar: it is assembling and compiling new code in memory.

Explaining the .NET CIL

In the world of computers there are two types of compiled programs: natively compiled programs, and Just-in-time compiled programs. Natively compiled programs are programs written in a language which, when compiled, are translated into instructions that can be understand by a targeted processor. **J**ust-**i**n-**t**ime programs, on the other hand, when compiled cannot be understood by the processor. Instead of compiling the program's code into native instructions, the JIT compiler translates the code into a secondary language, often called an intermediate language. Once the program has been compiled into an intermediate language, it is then ready to be distributed to users.

Because the program written in intermediate language can't be understood by the user's processor, it requires the assistance of what is known as a runtime interpreter. The purpose of the runtime interpreter is to read the program, and translate its intermediate language into instructions that can executed by that user's specific processor.

This is how programs written in C# operate. When a C# program is compiled, the compiler translates the code into a secondary language called the .NET **C**ommon **I**ntermediate **L**anguage. This code is embedded inside an EXE file, and the compiler switches a bit inside the file that causes the Windows OS to recognize the program as a program written in the .NET CIL. When the user tries to run the program windows reads the file, recognizes the bit that indicates its written in .NET CIL and passes the file to the .NET runtime. The .NET runtime then translates the CIL into instructions that that can be executed by the processor.

What is the benefit of one type of compilation over another? With native compiled programs, the software developer doesn't have to worry about the user having the runtime installed on their computer. Additionally, natively compiled programs are theoretically faster because they don't require the overhead of compiling the program as it runs.

JIT compiled programs have a few advantages of their own however. Because the program is compiled on the user's computer, the runtime executing the program can take advantage of specialized processor instructions that may be available. Natively compiled programs on the other hand are generally compiled to target the widest possible audience, often this means that they don't utilize instructions available on newer processors because not enough processors support it, and they don't desire to create multiple versions of the program that could confuse their userbase. Additionally, the developer only has to release one version of their program, and as long as there exists a runtime for the user's processor, the program will work. Whereas with natively compiled programs, you will often see x86, x64 and various other versions of a program that target the different processor architectures.

Fun fact: remember the VM from the previous section? I could alternatively describe the VM as a JIT compiler. There is not much of a difference between VM obfuscation and JIT compilation in this situation.

Assembling code in memory

Now, even though a JIT program is compiled as it runs, the job of assembling and translating code is entirely the job of the .NET runtime. It's very uncommon for a program to go out of its way to assemble and compile code on its own. It's also an excellent indicator of the developer hiding something.

Hashed name	Instruction
0x2A1EFD39	conv.ovf.i4
0xF0C9A0D0	refanytype
0x971E39F1	tail.
0x6C204D9B	ble.un.s
0xB9CDF884	bgt.s
0xC8C59EFD	ldelem.r8
0x762FDD69	ldc.i4.6
0xABE639B8	arglist
0x021DB37C	ldc.i4

Table 3

Examining createMethod, I find it is creating a function that consists of 38 instructions. All of those instructions, except two, are hardcoded into the program. Meaning that the program will always emit the same 36 instructions. The other two instructions are, however, determined from the array that was processed from the first calibration code. At the beginning of *createMethod* is a block of a code that requests a list containing the names of all possible .NET instructions. Those names are then hashed, and stored inside a table with their un-hashed equivalents (table 3). This table is then used, along with the array, to emit two instructions. To emit an instruction, *createMethod* reads a number from the array and looks that number up in table 3. If it finds a result it inserts the corresponding instruction.

If the wrong instructions (or no instruction) are given, the program will continue to emit instructions and compile the code as if the correct instructions were given. However, the code generated by *createMethod* will be corrupt. When the code is run, the .NET runtime detects a corrupt program and raises an exception. This exception will be caught by *CalibrationCodeIsValid*, indicating to it that code generation was not successful, and that the first calibration code is invalid.

With this information, the puzzle is revealed: I must determine which two instructions are missing. Once I've determined what the missing instructions are, I must use the hashed value of the instructions in the first calibration code. There are about 234 instructions in the .NET common intermediate language, so this task seems quite an effort. Thankfully, when *createMethod* is emitting the two instructions, it does it in such a way that prevents it from emitting any instructions that require operands (parameters). This detail means that the valid list of possible instructions is severely cut down.

Looking at the first missing instruction, I find that it is basically a freebie. The missing instruction is located right before an instruction that tells the .NET runtime to call a function called *get_chars*. This is a function that is called on a string. However, no strings are loaded before the call happens. In fact, the only string in the generated function is from a parameter it accepts. This tells me that the code is missing a "ldarg.0" instruction. This instruction loads the first parameter.

A few moments later, I determine that the second missing instruction is a multiplication instruction. I'll describe how I discovered this later.

Using table 4, I wrote the hashed version of the opcodes: "0364ABE72D29C96C". Entering this into the first calibration field causes no errors, and results in the code in figure 12.

Figure 12. The end product of the code assembled in memory.

```
static ulong dynamicFunc(string input_string)
{
    ulong num = 3074457345618258791UL;
    for (int i = 0; i < input_string.Length; i++)
    {
        num += (ulong)input_string[i];
        num *= 3074457345618258799UL;
    }
    return num;
}
```

The second calibration code

Looking at how calibration code 2 is used (fig 13), I notice that *CalibrationCodeIsValid* simultaneously loops through two arrays: the second calibration code, and a built-in array called “ASCII_Array”. As it loops through these arrays it takes a character from each array and appends them together, forming a two-character string. This string is then passed to the code generated in *createMethod*, which returns an 8-byte number. Then, *CalibrationCodeIsValid* checks if the returned number exists in a table called “Unknown_table”. If it does, the corresponding number is read from Unknown_table and then written, along with the character that was used from the ASCII_Array, to a table called “Table_generated_from_calibration_codes”.

```
// This block uses calibration code 2
for (int i = 0; i < ASCII_Array.Count; i++)
{
    string input;
    if (i < calibration_code_2.Length)
    {
        input = ASCII_Array[i].ToString() + calibration_code_2[i].ToString();
    } else
    {
        input = ASCII_Array[i].ToString();
    }
    var result = dynamicFunction(input);
    if (Unknown_table.ContainsKey(result))
    {
        Table_generated_from_calibration_codes.Add(ASCII_Array[i], Unknown_table[result]);
    }
}
```

Figure 13. Code that uses the second calibration code

ASCII_Array	Unknown_table	
'0'	0x04BDA12F68E4A01E	2052
'1'	0xB8E38E38E38EF81C	10
'2'	0x212F684BDAA0A6AD	32
'3'	0xDA12F684BE368A63	2304
'4'	0xE38E38E38E39D18F	1537
'5'	0x212F684BDAAB012D	2056
'6'	0x0E38E38E38E45CC8	1026
'7'	0xE38E38E38E39C3C1	2562
'8'	0xE38E38E38E3995BD	578
'9'	0x0E38E38E38E46A96	2624
'A'	0xBDA12F684C754FAD	1152
'B'	0x0E38E38E38E44EFA	518

Table 4. The data contained in the ASCII_Array and Unknown table

From the data in table 4, I can see that the first character of the calibration code will have ‘0’ as a prefix, the second will have ‘1’, and so on. The first column of Unknown_table contains numbers that are matched against the output from the *createMethod* generated function. From this data, I can determine that the generated function is basically acting to hash the strings. This is why I realize that the second missing instruction is a multiplication instruction. If we look at figure 13, we can see that it takes a large eight-byte number, and adds the value of each character from the string to that number. Looking at the first column in Unknown_table, we can observe a very large range of numbers. Multiplication is the one of the few instructions that could produce the wide gamut of numbers seen in Unknown_table from the code in figure 13.

At this point, the puzzle looks like this: I need to find a string that, when used alongside the characters from ASCII_Array, will produce numbers that match the first column of Unknown_table. To do this I create a python script that implements the code from figure 13. I then copy the data from ASCII_Array and Unknown_table data into the script.

With these two arrays together, I setup the script to loop through every ASCII character possible, appending it to the first character from the ASCII_Array. These combinations are then passed to the hashing algorithm, and the result is checked if it exists in the Unknown_table. If it does, the script makes note of what character produced the successful check, and continues onto the second character in the ASCII_Array. Basically, it passes the hashing algorithm these strings:

Next, I setup the script to do the following:

- For every character in the ASCII_Array run a loop that takes that character and appends to it another ASCII character, resulting in a two-byte string.
- Pass the two-byte string to the python version of the code in figure 13.
- Check if the result is found in the Unknown_table data.
- If a result is found, record the character that was appended to the ASCII_Array character. Next, start the loop process all over again but on the next character in the ASCII_Array.
- If no result is found, keep trying different characters until there are none left to try.

Because there are 95 printable ASCII characters, loop would run 95 times before running out of characters to try. In the worst-case situation it would take 5,320 loop iterations before the script would finish (95 loops on each character * 56 characters in ASCII_Array). Running the script, it takes only a quarter of a second for the script to find a match: “Infant Jesus of Prague”. And not only does entering that text into the second calibration code field cause a new UI to appear, but entering this text into the third password field of the ESET Crackme shows that the text is our third and final password! One final mystery remains: the punch card reader.

The Punch Card Reader

The final UI of the Puncher Machine contains a large textbox with a button that says “punch it”. Entering any random text and clicking the button causes the Puncher Machine to play sounds as it “punches” the text into a series of punch card images.

Because of prior examination of the Punch Card Reader when I was trying to figure out the source of the punch card files, I already know that the Punch Card Reader program has a similar puzzle consisting of a *createMethod* function that assembles and compiles code. In this version it is missing three instructions. The punch card reader deviates however in that the instructions aren’t hashed, and instead the program just reads the instruction names from the punch card images. The text is then used to match against an instruction.

```
static bool codeFromPunchCardReader()
{
    uint num = 57005u;
    uint num2 = 48879u;
    uint num3 = 51966u;
    uint num4 = 47806u;
    uint num5 = 64206u;
    uint num6 = num / num2 ^ num3 / num4 ^ num5 ^ 4065355188u;
    byte[] bytes = Encoding.ASCII.GetBytes("ESET");
    uint num7 = BitConverter.ToUInt32(bytes, 0);
    return num6 == num7;
}
```

Figure 14. The code generated from Punch card reader. Missing instructions have been replaced with division instructions.

Examining the code that the program emits, it appears that this one is a freebie by the ESET developer. The code roughly does the following: it takes a series of numbers, manipulates them, and compares them to the string “ESET”. The numbers are already built into the emitted code, and the only thing I need to determine is which math instructions should be run on the numbers. In figure 15 I’ve replaced the missing instructions with division instructions, just to get a view of the code.

I decide to write a python script that implements the same code as in figure 14. This script will run the code in figure 14 over and over with different math instructions (in place of the circled divisions) until a combination is found that causes the code to return true.

The script quickly finds a match, replacing the first division with a multiplication, and the second division with an addition. The third and final instruction is free, because its located at the very end of the generated code. The final instruction is always a ret (return) instruction. Printing out a series of punch cards with the mul, add, ret instructions and passing them to the punch card reader causes a message dialogue to pop up congratulating me, and the project is done!