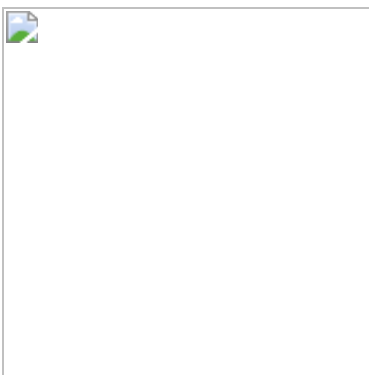


TangleCrypt: a sophisticated but buggy malware packer

 labs.withsecure.com/publications/tanglecrypt



by **Bert Steppe**

Strategic Threat Intelligence & Research Group (STINGR)

27/11/2025

Executive Summary

WithSecure's STINGR Group is releasing a detailed technical analysis of TangleCrypt, a previously undocumented packer for Windows malware. The packer was found on two executables used in a recent ransomware attack and their payloads were both identified as an

EDR killer known as STONESTOP that leverages the malicious ABYSSWORKER driver.

Key findings about TangleCrypt:

- The payload is stored inside the PE Resources via multiple layers of base64 encoding, LZ78 compression and XOR encryption.
- The loader supports two methods of launching the payload: in the same process or in a child process. The chosen method is defined by a string appended to the embedded payload.
- To hinder analysis and detection, it uses a few common techniques like string encryption and dynamic import resolving, but all of these were found to be relatively simple to bypass. The lack of any advanced anti-analysis mechanisms also makes manual unpacking of the payload rather straightforward.
- Although the packer has an overall interesting design, we identified several flaws in the loader implementation that may cause the payload to crash or show other unexpected behaviour.

Introduction

A popular technique among threat actors in hands-on attacks to evade detection, is the use of an EDR killer before conducting malicious operations like data exfiltration and file encryption in ransomware attacks. Such tool forcefully terminates the installed security products on the device, either by exploiting a vulnerable driver ([BYOVD](#)), or via a malicious driver specifically created for this purpose. The latter was found to be used in an attack from early September 2025, where the threat actor deployed Qilin ransomware on the victim's systems. WithSecure's Incident Response team recovered the following artifacts, and reached out to the Strategic Threat Intelligence & Research Group (STINGR) to investigate them in further detail:

- 4cBh.exe - an x64 executable, compiled using GCC/MingW. First packed with VMProtect, and subsequently with a custom executable packer that we are tracking as TangleCrypt.
- b1.exe - another x64 executable, with identical functionality and applied packers as 4cBh.exe, but compiled using Visual Studio.
- fehmr.sys - an x64 kernel driver, packed with VMProtect, and masquerading as a CrowdStrike Falcon Sensor driver.

We identified the kernel driver being ABYSSWORKER (older versions also known as POORTRY and BURNTCIGAR) and the payloads embedded in the executables being STONESTOP.

Technical details can be found in several write-ups (see References), but here is a quick summary of how the malware works:

- Upon execution, STONESTOP first checks if it has elevated privileges. If not, it will restart itself to run with admin rights, which triggers a UAC prompt by default.

- Once it knows that it has the elevated rights, it registers the ABYSSWORKER driver in Windows. The executable assumes that the SYS file is present in the same directory. Then it tries to load and initialize the driver.
- The executable contains a list of executable names and uses the driver to terminate all running processes matching an item in this list.
- Some STONESTOP versions also have a hard-coded list of directories. All files in these directories will be recursively deleted by the driver.
- The executable keeps running in the background. The termination of any running 'unwanted' process is repeated every second.

Although the ABYSSWORKER sample in the Incident Response(IR) case handled by WithSecure was not publicly known yet, its functionality is identical to the sample analyzed by Elastic. In fact, it is even the very same driver, only the packer is different: the sample Elastic described is protected (albeit poorly) with CodeVirtualizer, while the file in our case is protected with VMProtect, just like most ABYSSWORKER samples that we found elsewhere. (See Appendix B)

STONESTOP in-the-wild

During our investigation, we also hunted for more STONESTOP samples, as we were intrigued by the earlier observation that one sample targets different security products than the other. We were able to confirm this: all found executables target Microsoft Defender and usually one or two other EDR/AV vendors. There are a few exceptions - we found one STONESTOP sample that targets Defender only, and a few samples that target up to six other products. A detailed overview can be found in Appendix B.

But we observed something quite remarkable regarding the usage of executable packers on STONESTOP. All identified samples are packed with VMProtect, and sometimes additionally packed with a malware packer known as HeartCrypt and well documented by Palo Alto. (See References)

However, the two STONESTOP samples '4cBh.exe' and 'b1.exe' from our IR case looked different from a PE structure point of view. While HeartCrypt-packed samples typically have a lot of items in the PE resources and lack relocations, these two samples do have relocations but have only two resource entries.

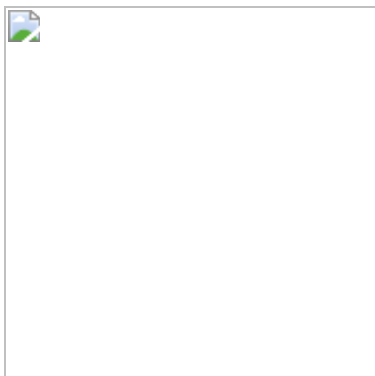


Figure 01A: Resource tree HeartCrypt

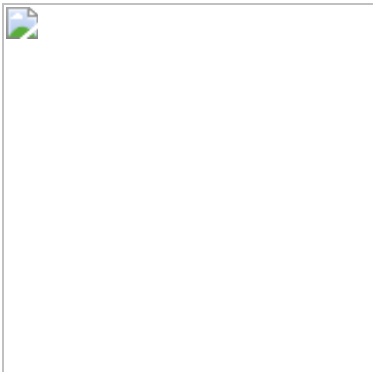


Figure 01B: Resource tree TangleCrypt

Our initial thought was that we may have found an updated version of HeartCrypt that supports x64 (because, according to the known advertisements, it only supported x86 and .NET), but a closer inspection revealed that the internal workings are so divergent from HeartCrypt that we concluded that we were dealing here with a completely different malware packer, which, to our knowledge, has not been reported before. Our suspicion was reinforced by the fact that we found only one other sample with similar packer characteristics, 73b6e7cdd10c373a633367fd3bde791278e7900b342a21e2bad2b8e5cfc33746, containing an XWorm payload.

We decided to name this packer “TangleCrypt”, for reasons explained below.

Meet TangleCrypt

Just like most malware packers, TangleCrypt's main objective is to hide the actual payload and make it look like a benign file. The original executable is encrypted inside an entry of the PE resources in the '.rsrc' section, while the rest of the file is mostly the loader code itself - this is rather atypical for malware packers, where often completely irrelevant code and data is inserted as an additional technique to make the file look more legitimate.

However, a more typical packer characteristic that TangleCrypt adopts, is the attempt at lowering the overall entropy of the sample to make the presence of an encrypted object less obvious. The '.rdata' section is full of 0x8D7-size areas which are 0x00 bytes but all encrypted with the same XOR key, resulting in the same byte pattern and thus having a low entropy.



Figure 02: Two regions of encrypted NULL bytes

Another interesting observation is that the loader code in both samples ('4cBh.exe' and 'b1.exe') is 100% identical; only the encrypted payload in the PE resources is different. However, TangleCrypt shows a significantly different behaviour in each sample. In '4cBh.exe', the original payload is decrypted in its own process memory, while 'b1.exe' starts a child process of itself and writes the decrypted payload to that process. How this is achieved, is explained in detail later in this blogpost.



Figure 03: ProcessMonitor log of 'b1.exe' starting child process of itself

Anti-analysis tricks

TangleCrypt uses some techniques to make analysis more challenging, but they are limited and can be worked around relatively easily. Here is an overview.

Anti-debugging

The TangleCrypt authors did not really try to prevent manual dynamic analysis with a debugger. The only small trick we identified, is that the loader calls two Win32 API functions [LocalHandle](#) and [GlobalSize](#) with the first and only argument set to NULL, which is an invalid value for both API functions. This triggers an Access Violation exception, but will be caught by the exception handler in the API itself.





Figure 04A/B: Disassembly of LocalHandle and GlobalSize calls

In practice this means that the debugger will receive a "first-chance" notification for the two exceptions, and the debugger may halt the execution. Bypassing this is trivial: either we simply let the target continue the execution, or we configure the debugger to ignore Access Violation exceptions in the current debug session.

Since it's so easy to bypass in a debugger, we rather suspect that the authors included this trick as a countermeasure against code tracers - for example used in some automated sandboxes - since an exception might confuse them.

String encryption

To hinder static analysis, lots of strings are not stored in plain sight, but rather in encrypted form, either fetched from the '.rdata' section or constructed at runtime on the stack. Although all strings are encrypted using the same algorithm, there are dedicated decryption functions (over 100) for most encrypted strings, which makes it more challenging to analyse the binary as we cannot set a "central" breakpoint in the code to print the decrypted string.



Figure 05: Disassembly of one of the many functions that decrypt a fixed string. The encrypted string is constructed on the stack, followed by a loop that decrypts the string character per character.

Or can we?

When taking a closer look at the decryption loop, we can see that it calls function 'sub_140010870' for every byte it wants to decrypt. This function calculates a division value based on the string length, and is called in all string decryption functions. Another interesting fact is that all these functions store a pointer to the "string-being-decrypted" at the same relative position on the stack: '[rsp+30h]'. We can use this to our advantage to set a breakpoint at the beginning of 'sub_140010870' and there print the string referenced at '[rsp+38h]' (we need to add 8 because the 'call' instruction pushed the return address on the stack). For example in WinDbg we can enter the following command, which will set a breakpoint at sub_140010870, and when hit, it will print the ANSI string and continue the execution:

```
bu 140010870 "da poi(@rsp+38);g;"
```

(assuming the exe is loaded at address 0x140000000)

When we run the executable, WinDbg shows the following output:

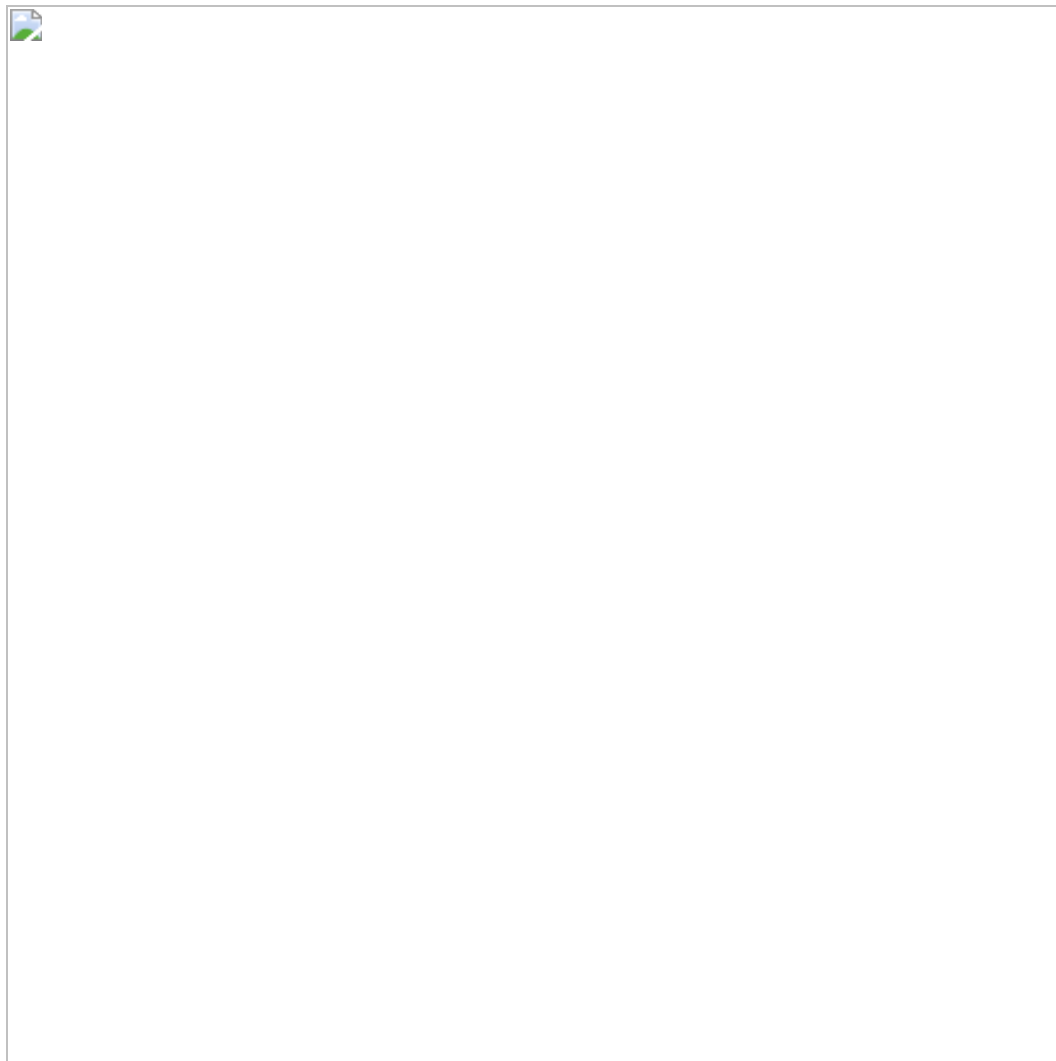


Figure 06: WinDbg

console output. Here the strings ".text" and "kernel32.dll" are being decrypted.

As you can see, the output is a bit rudimentary, as it will print the string for each character being decrypted, and we don't see the last character of the decrypted string because the function is not called after decrypting the last character. But it does provide us a good insight into the hidden

strings used by TangleCrypt, especially given the simplicity of the approach.

Dynamic import resolving

To prevent that the PE imports reveal much of its actual capabilities, TangleCrypt uses the string encryption feature also to decrypt strings of Windows DLL names and API function names, which are then used to dynamically resolve their virtual address.

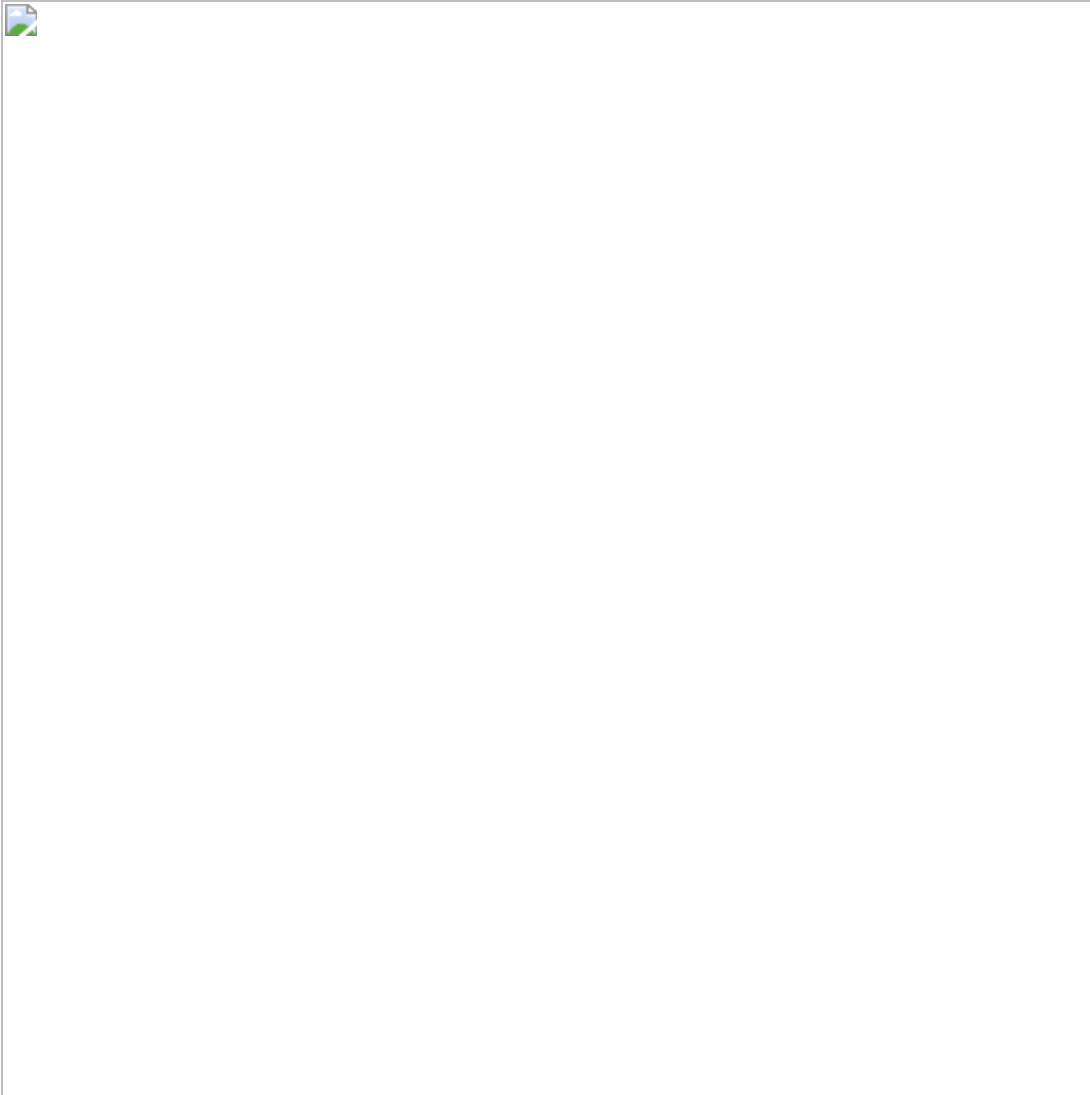


Figure 07: More

WinDbg console output. The loader is clearly using functions to read PE resources, but this cannot be determined via static analysis.

Unoptimized code

A final interesting observation is that the TangleCrypt loader has been compiled without any code optimization. The consequence is that the loader code is somewhat bloated, making it more time consuming to trace through with a debugger. It is unclear whether this is even intentional or if the authors disabled code optimization for a technical reason.



Figure 08: Example of unoptimized code: redundant jumps to the next instruction, and values written to local variables on the stack while they could be just copied between registers.

Payload decryption & different behaviour

This section describes how the payload decryption happens - which is more complex than what we expected.

First, TangleCrypt resolves a few resources related functions in KERNEL32.DLL via the string decryption routine: (see Figure 07)

LoadResource
SizeofResource
etc

As mentioned before, the executable payload is stored in the PE resources. However, there is another very small entry to be found in the resources, and that's exactly what the loader fetches first via these API calls.

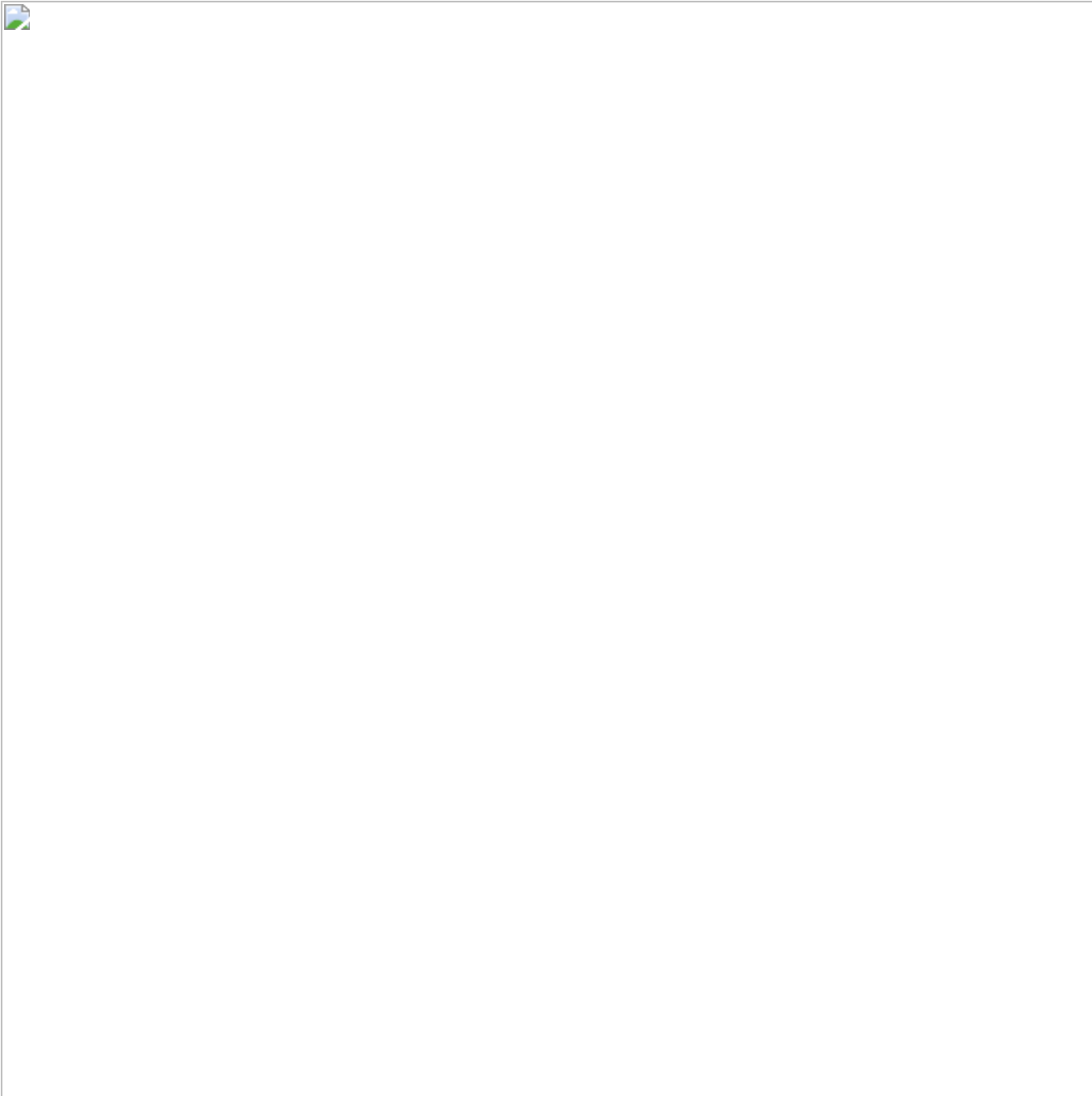


Figure 09: Fetched "small" resources entry

This is a base64 encoded string, which the loader decodes.

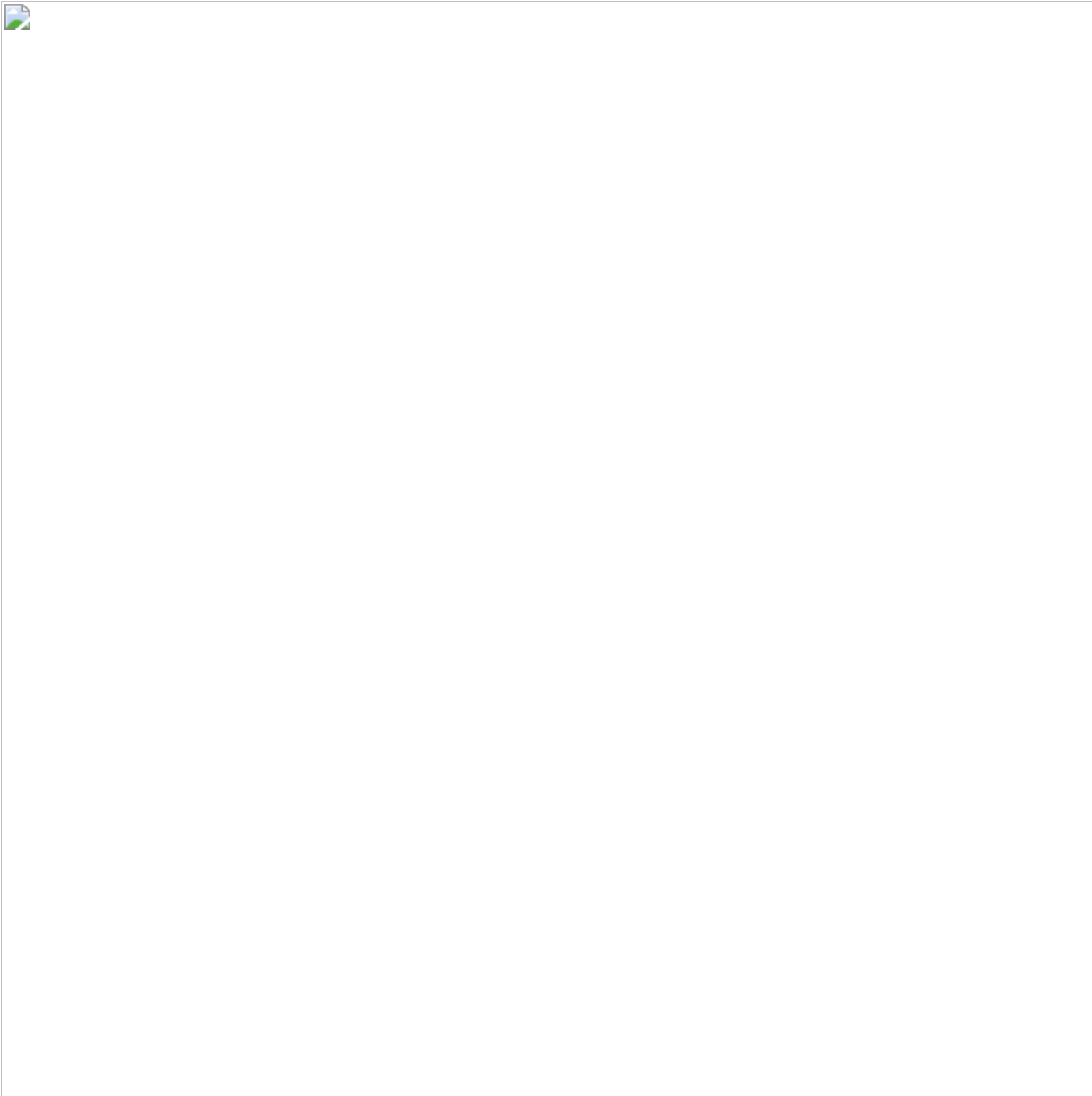


Figure 10: base64 decoded string

The output is less obvious to recognize. It looks like a sequence of numbers but with some NULL bytes in between. It turns out that this is an LZ78 compressed buffer, and the loader calls a function to decompress it.

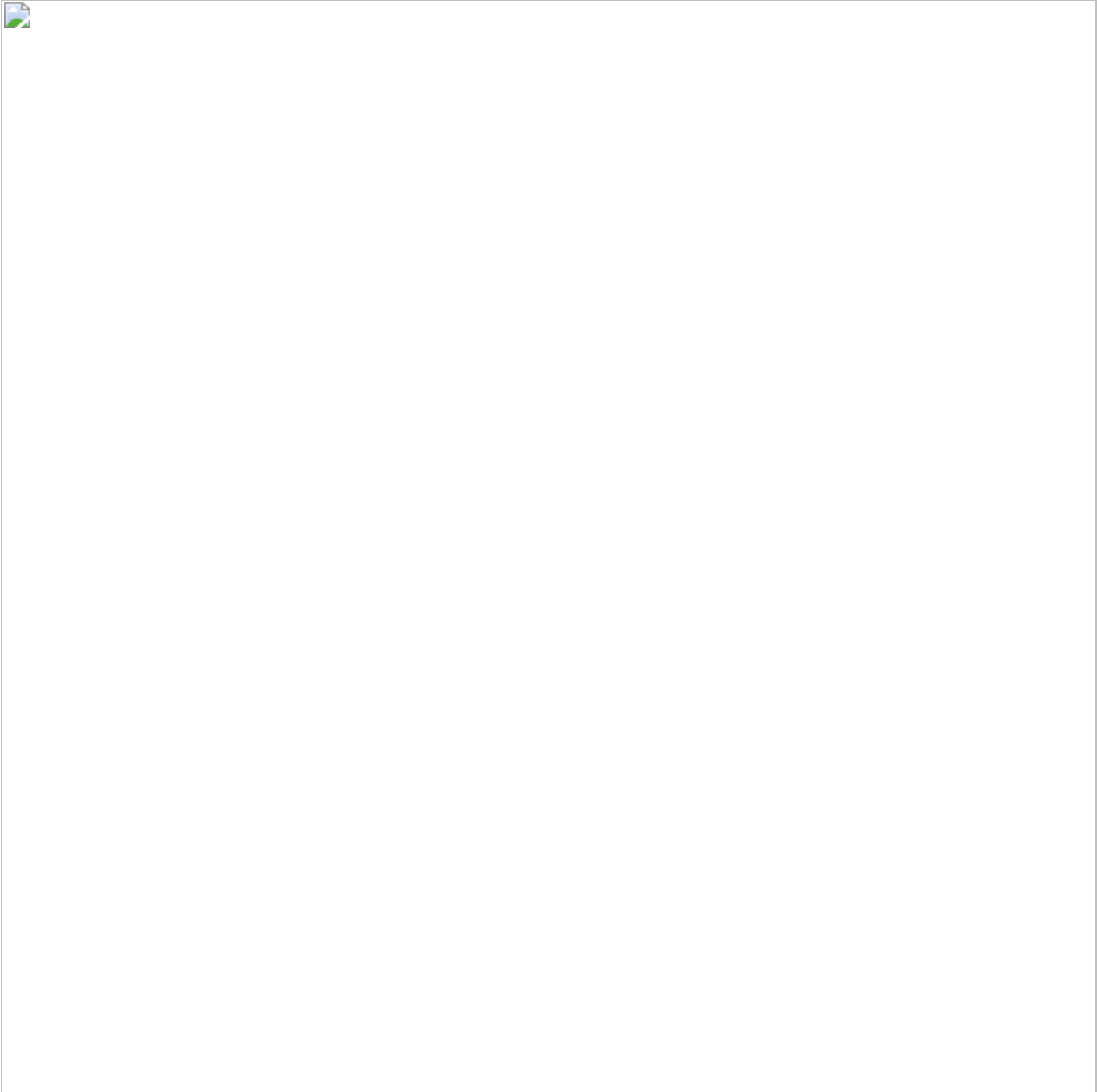


Figure 11: LZ78 decompressed buffer

Since the original data buffer is so small, there was little to be compressed; in fact, the compressed buffer is larger than the decompressed buffer, which is only 6 bytes. Anyway, we end up with a string consisting of 6 numbers, in this case “175438”.

Next, the loader fetches the other PE resources entry, which is significantly larger.



Figure 12: Head of fetched "large" resources entry

Another base64 encoded string, and after decoding we have again a buffer whose structure looks somewhat familiar.

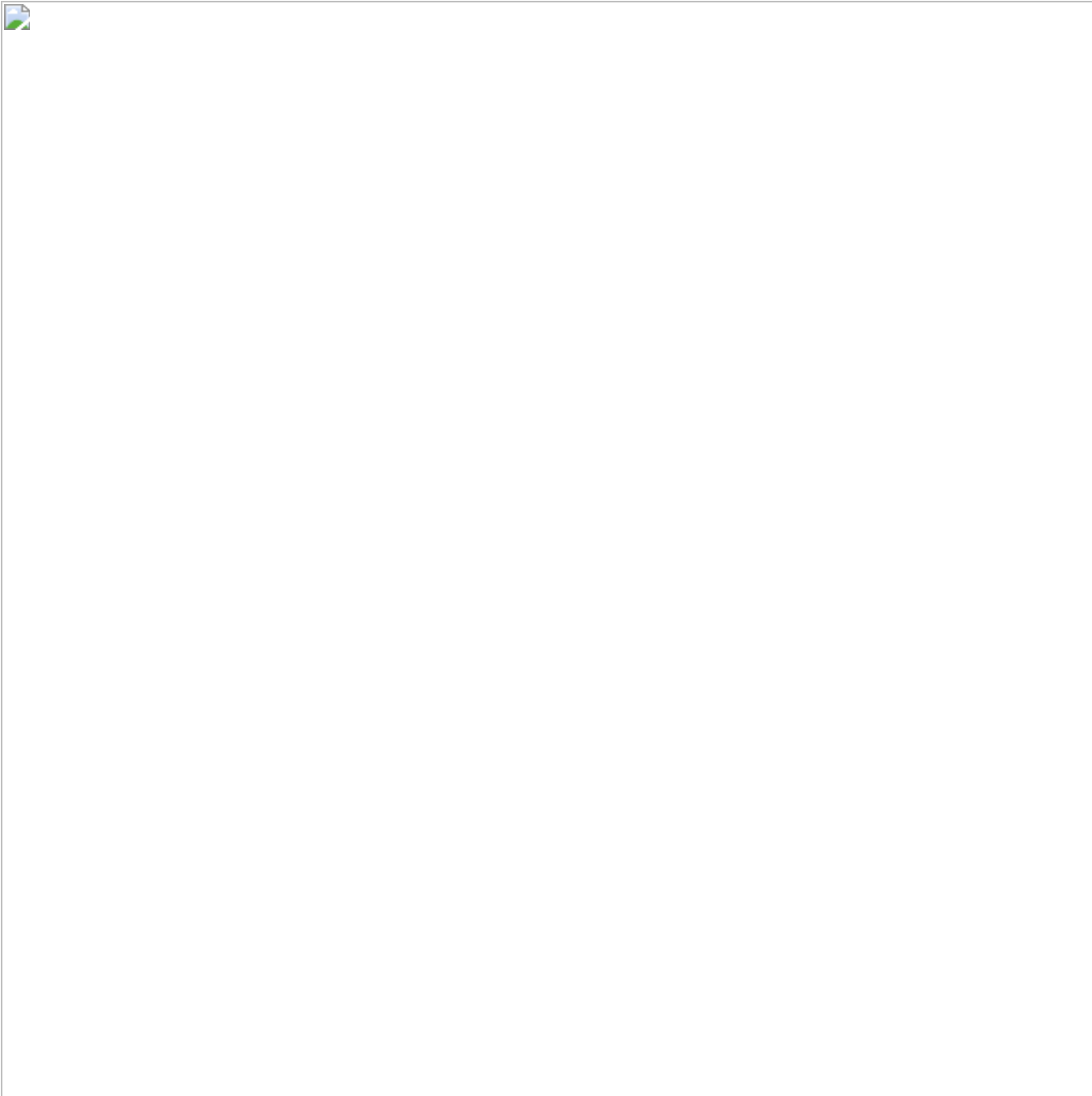


Figure 13: Head of base64 decoded string

Indeed, another LZ78 compressed buffer. But this time it decompresses into a larger buffer.

Note: due to the unoptimized assembler code, decompressing this buffer takes a few seconds even on a decent CPU.

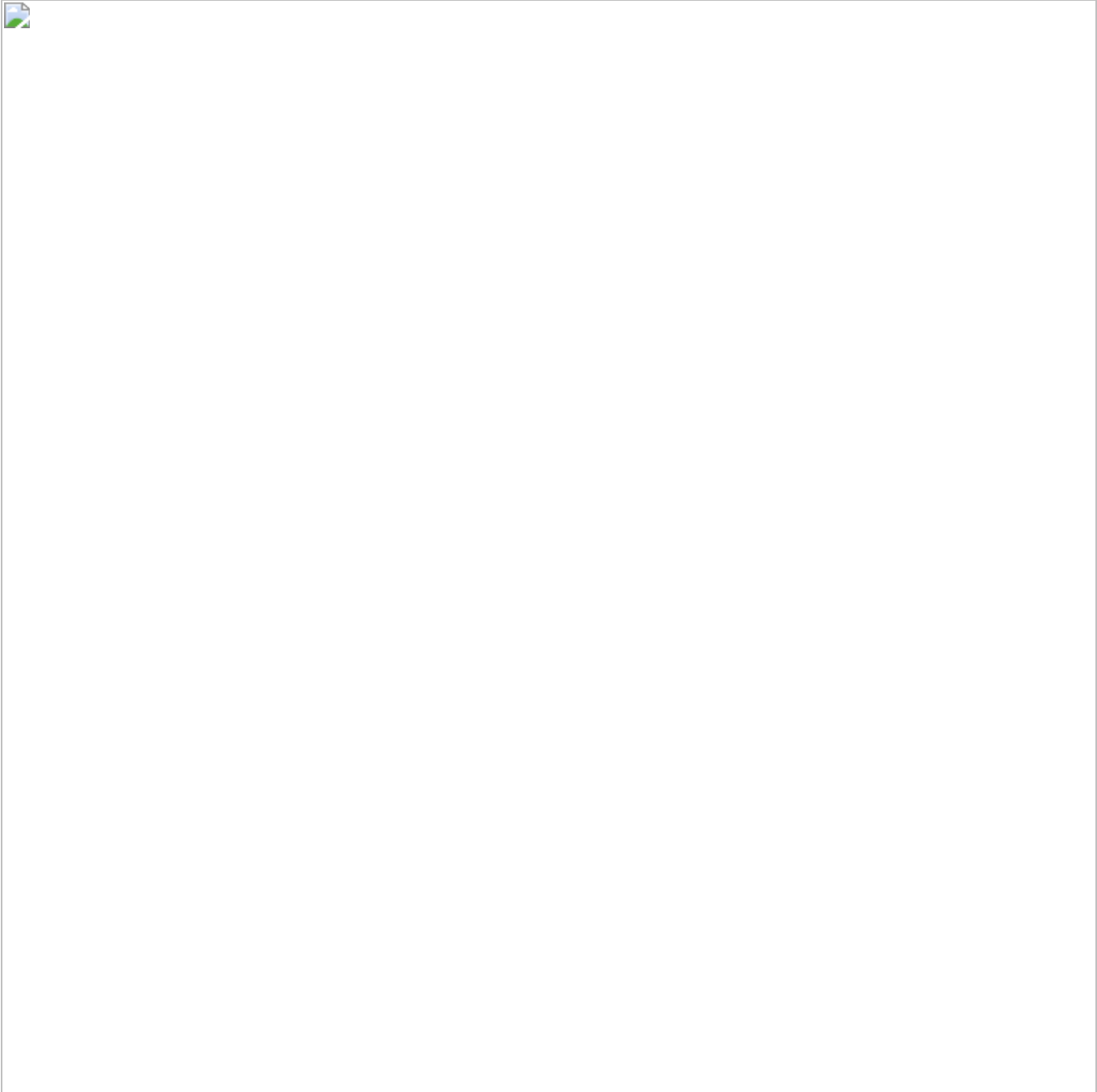


Figure 14: Head of LZ78 decompressed buffer

The result is again base64 encoded data, which the loader of course decodes.

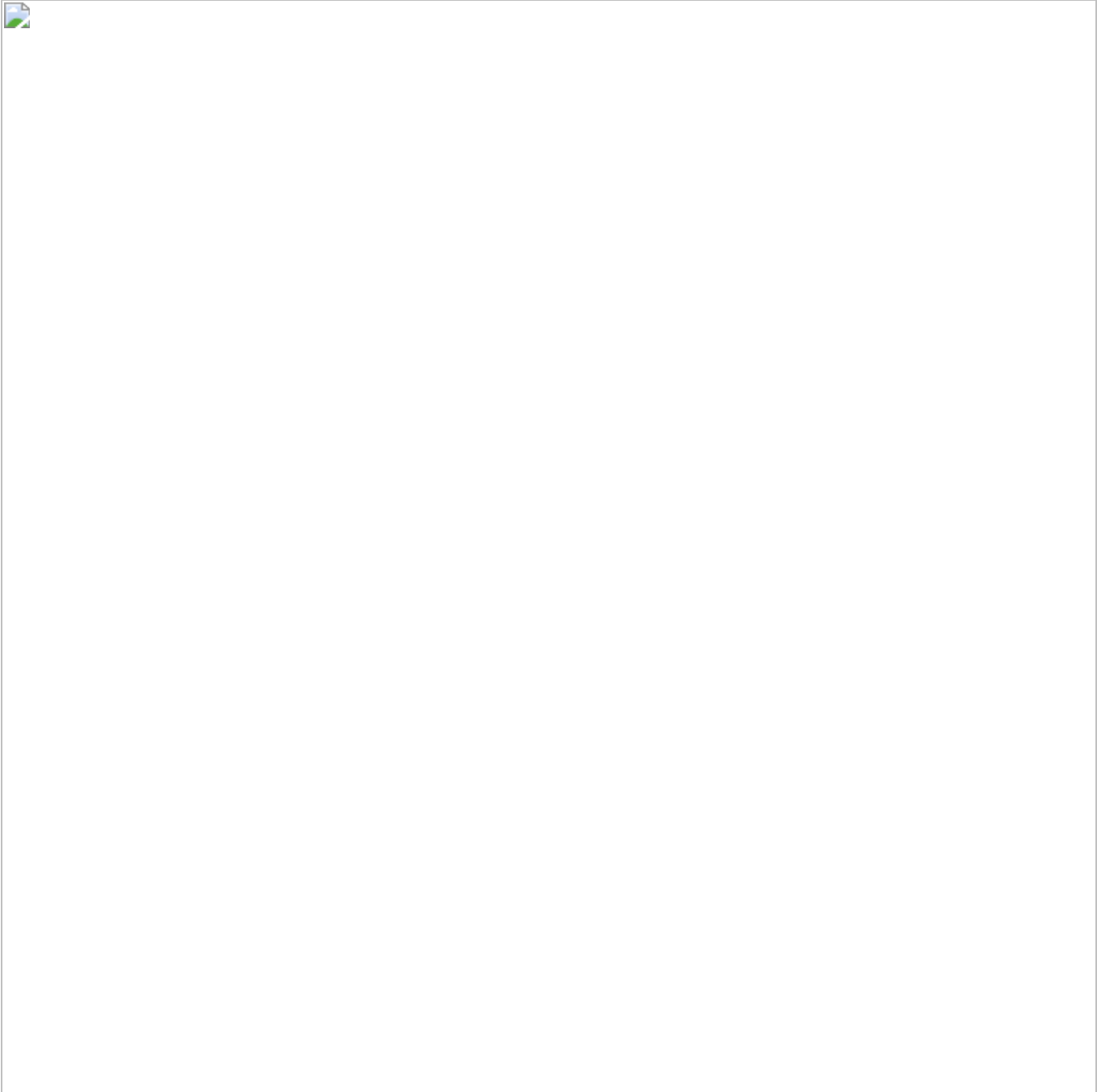


Figure 15: Head of base64 decoded data

If some byte sequences look similar to something we have seen before, that's normal. The loader uses the number sequence - decrypted from the small PE resource - as a XOR key to decrypt this buffer.

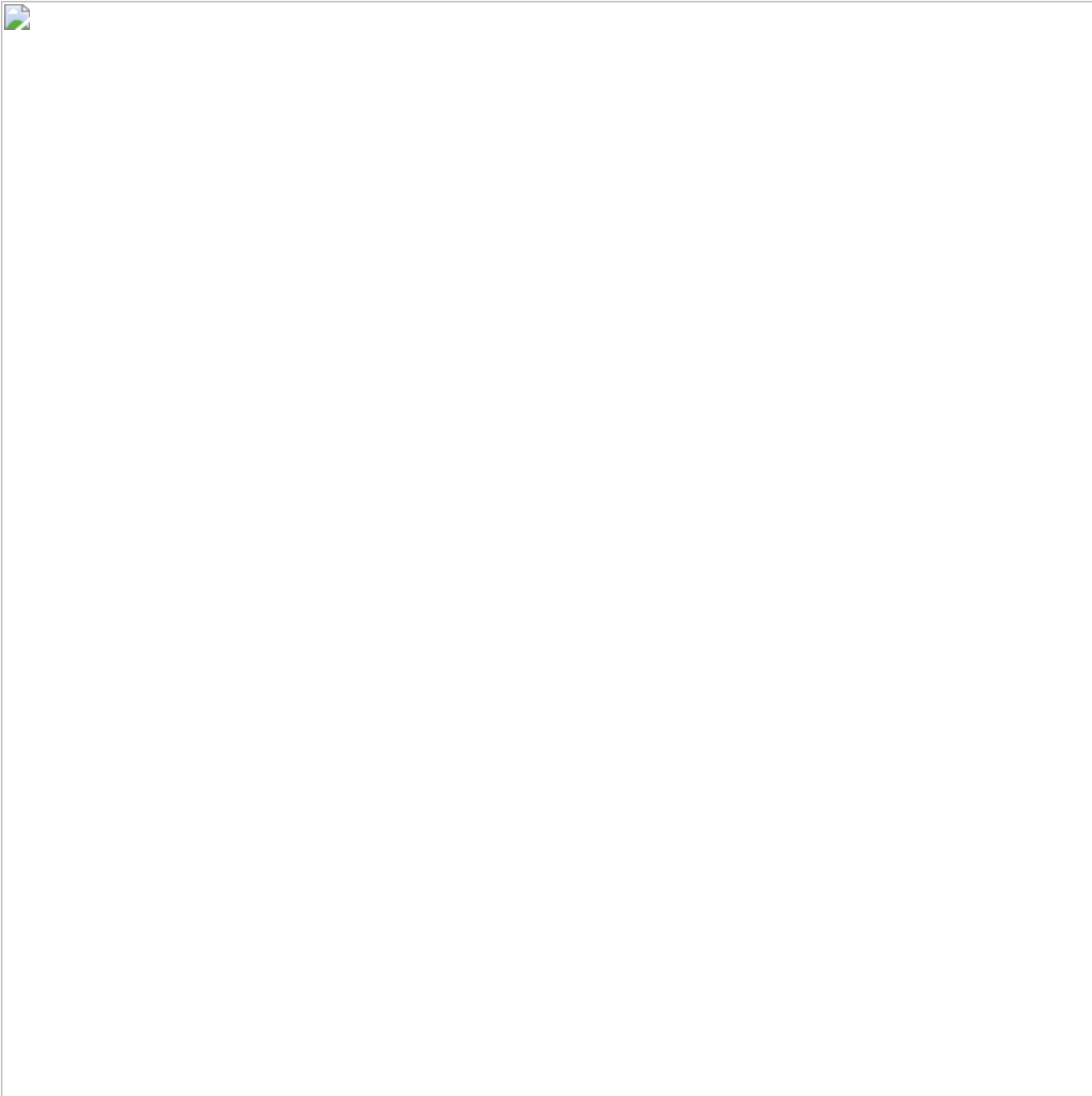


Figure 16: Head of decrypted buffer
And this is finally the original executable.

Now let's come back to an earlier observation. '4cBh.exe' decrypts the payload in its own process, while 'b1.exe' starts a child process where the decrypted payload is written to. The loader code is identical, so how does it know what to do? The answer can be found at the very end of the decrypted payload:

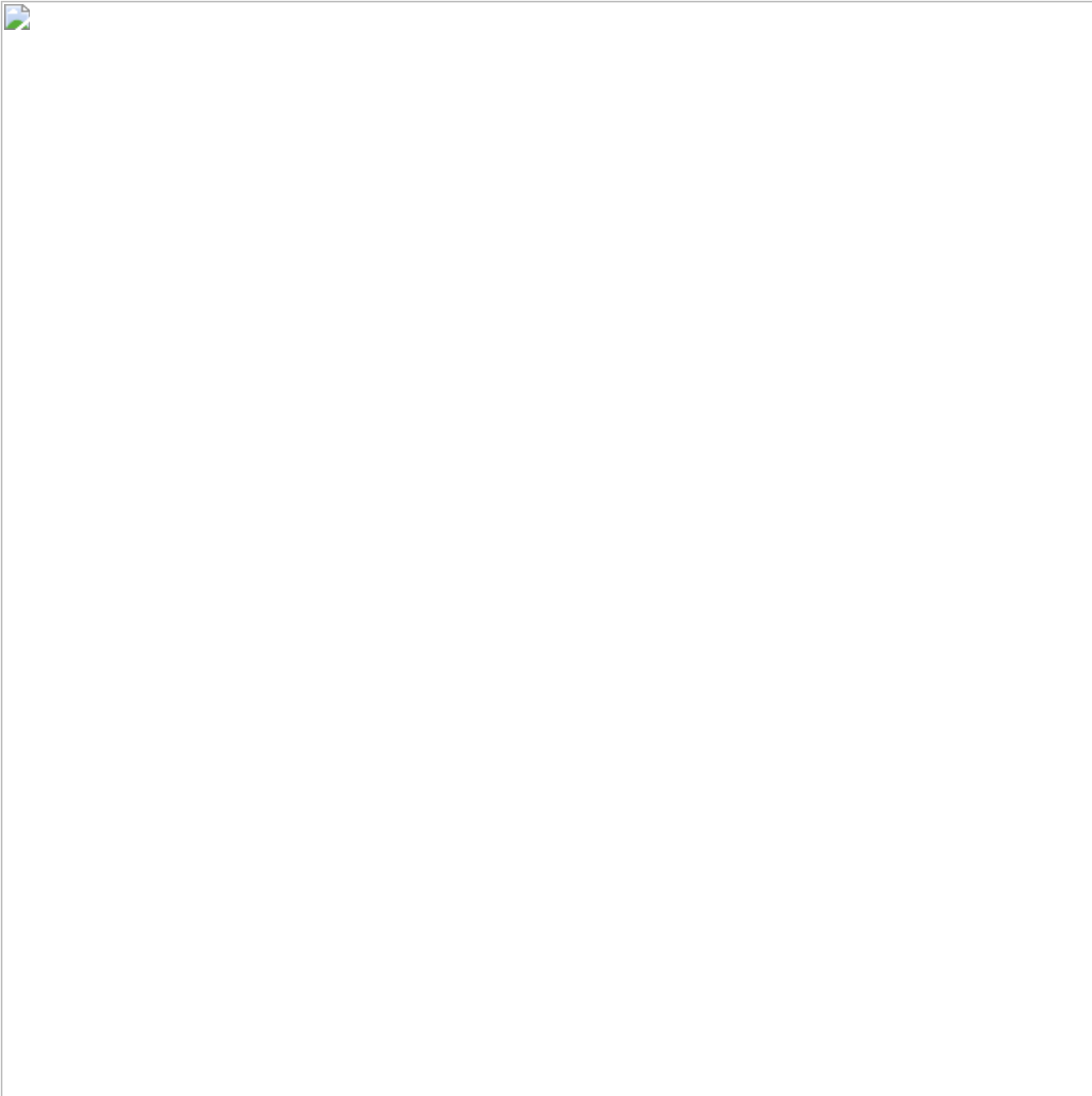


Figure 17: Tail of decrypted buffer

There is a special string that instructs the loader how to process the payload, and that is different in the two samples.

File	String	Behaviour
4cBh.exe	exex64_amd64_block_	inject in own process
b1.exe	exex64_amd64__riin	inject in child process

We were able to confirm this behaviour by swapping the strings in the process memory immediately after the payload was decrypted: when we changed the string in '4cBh.exe' into "exex64_amd64__riin", a child process was created, and when we changed the string in 'b1.exe' into "exex64_amd64_block_", the payload was executed in the same process.

The way of launching the payload is most likely a builder option that the threat actor can choose from when they generate the packed sample.

Manual unpacking

Suppose that we have several PE files packed with TangleCrypt and that we just want to unpack them all without having any pre-knowledge of the internal workings of the loader. Tracing through the code is always an option, but time consuming - we would like to find a quick and easy way to dump the unpacked executable to disk.

Contrary to many commercial protectors, malware packers usually don't modify the original PE to make it more difficult to produce an unpacked executable: the original file is present in the process memory at some point, it's just a matter of halting the target process at the right moment. TangleCrypt is not an exception here, but the approach depends on the way it processes the payload.

4cBh.exe

If TangleCrypt executes the payload in its own process, the loader calls KERNEL32.DLL function [VirtualAlloc](#) to allocate the memory where the PE will be prepared for execution. If we launch the process and set a breakpoint on this API (in WinDbg: "bu kernel32!VirtualAlloc"), the target will hit the breakpoint after a few seconds:



Figure 18: WinDbg disassembly + registers when hitting the VirtualAlloc breakpoint
VirtualAlloc has the memory starting address as first argument ('rcx' as per the calling convention in x64) and the memory region size as second argument ('rdx'). So in this case, a memory region of 0x24E000 bytes will be allocated at address 0x400000.

Let's step over this function and have a look at the code that calls it:



Figure 19:

WinDbg disassembly of code that calls VirtualAlloc

Here we can see that the loader gets the allocation size from a memory region that the QWORD at '[rsp+120h]' points to. What can we see there?

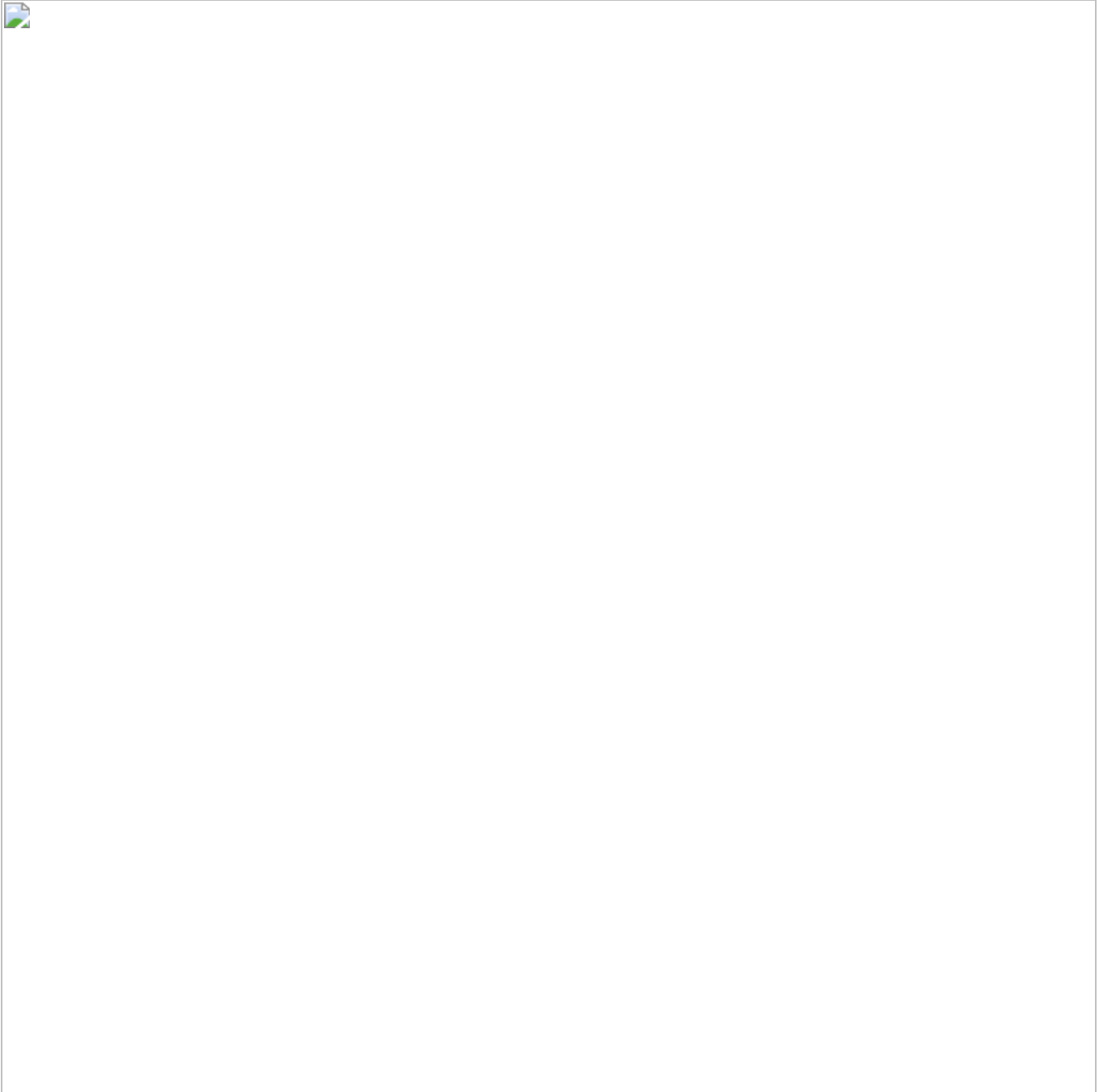


Figure 20: Process memory

This looks like a PE header. The TangleCrypt loader gets the PE image size from it, which makes perfect sense. And if we scroll up a bit, we also find the MZ header, the start of the original unpacked executable.

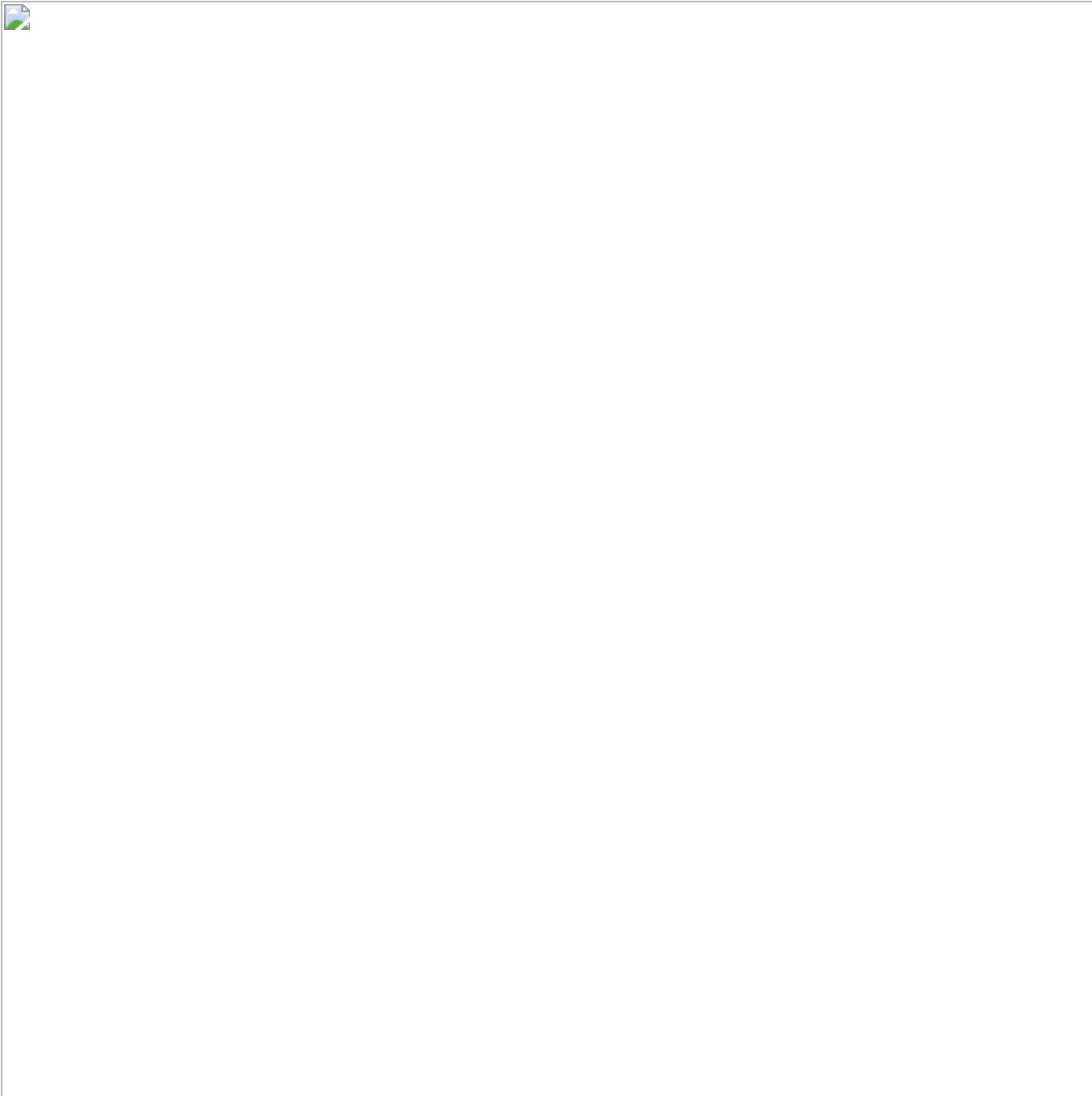


Figure 21: MZ/PE header

Dumping this memory to disk results in a perfectly valid PE executable that can be launched right away.

b1.exe

In the other sample, the loader starts a child process of itself in suspended state, writes the decrypted content to the child process memory and finally resumes its main thread. TangleCrypt uses the most straightforward way to achieve this, via the Win32 API functions [CreateProcessW](#), [WriteProcessMemory](#) and [ResumeThread](#) in KERNEL32.DLL.

Let's start the target process and set a breakpoint on WriteProcessMemory. After a few moments, the process will break:



Figure 22: WinDbg disassembly + registers when hitting the WriteProcessMemory breakpoint
WriteProcessMemory's second argument ('rdx') is the memory address in the target process where the memory will be written to, and the third argument ('r8') points to the buffer to write. What is this buffer?

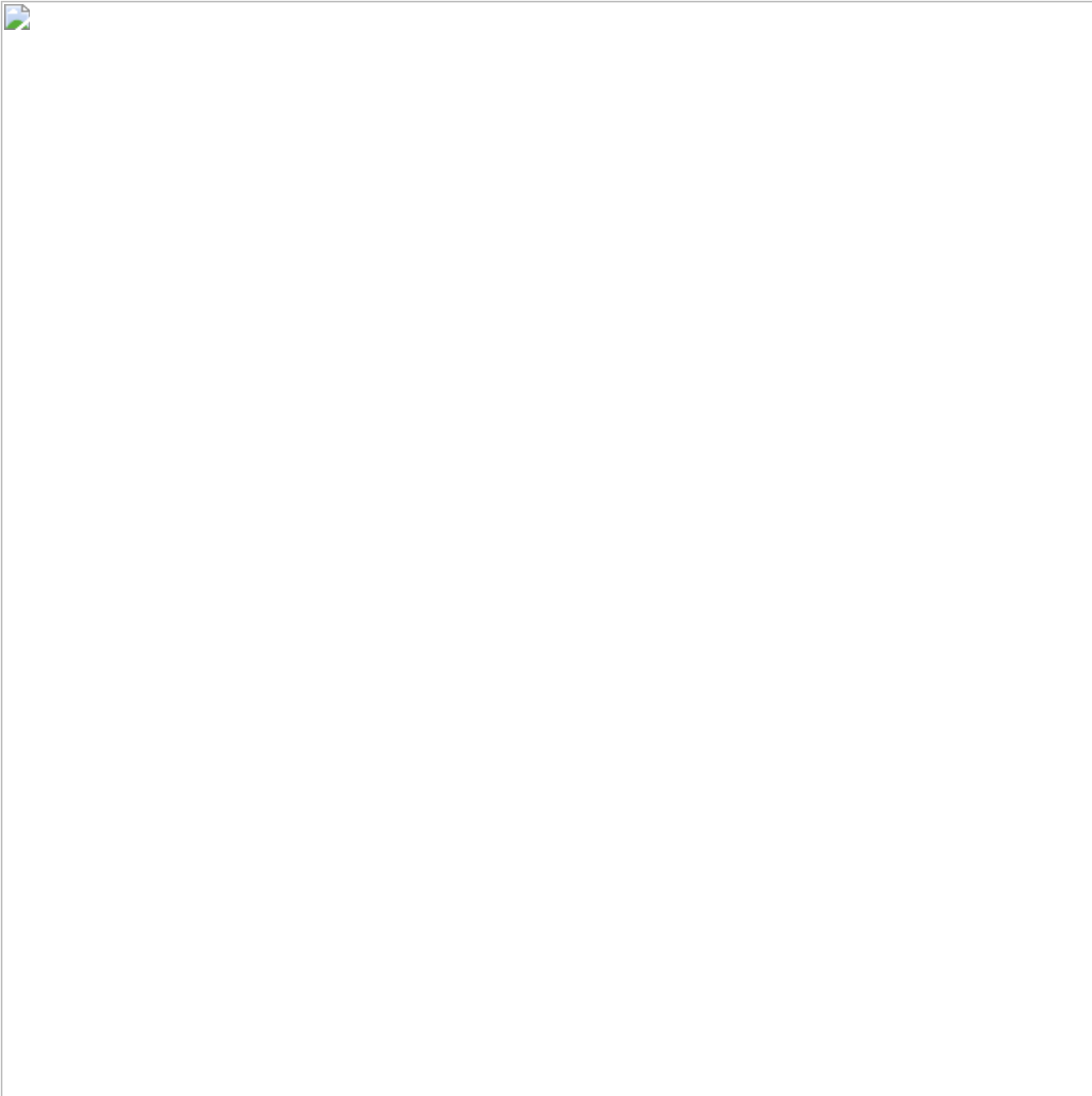


Figure 23: Process memory

Here we immediately find the MZ/PE header of the original file. Dumping this memory region to disk is again all we need to do to have a working unpacked executable.

Glitches

When we started the analysis on the STONESTOP executables from this IR case, we noticed that one of the samples - '4cBh.exe' - crashed if we executed it without admin rights. Initially we thought that this was due to a bug in the STONESTOP logic itself, but when we later discovered that the unpacked sample did not crash and displayed the expected UAC prompt, we realized that the crash must be caused by TangleCrypt instead.

Identifying the problem

As long as your debugger has not been configured to completely ignore Access Violation exceptions to bypass the previously mentioned anti-analysis trick, it easily reveals that the crash happens in MSVCRT.DLL function 'strchr'.



Figure 24: Disassembly of the code where the crash occurs

This function returns a pointer to the last occurrence of a character (second argument, 'rdx') in a given string (first argument, 'rcx'). But as you can see, 'rcx' is 0 here, [and strchr does not validate its input](#); hence the null pointer exception.

Let's take a closer look at the code that calls this 'strchr' function.



Figure 25: Disassembly of STONESTOP code

This code is part of the original STONESTOP '.text' section (virtual address 0x401000 - 0x405000), so at this point, both TangleCrypt and VMProtect have unpacked their payload.

At 0x401FC6 it calls a function that basically checks if the process is executed with admin rights. If that's the case, it returns 1 in 'eax', otherwise 0, where it jumps to the code at 0x401FDE that prepares a few things to start itself with elevated rights.

The next relevant instruction in this logic is at 0x401FE7: it fetches a pointer to MSVCRT.DLL export [pgmptr](#). This is not a function; it is a pointer to a variable that contains a pointer to the full path of the executable (or the executable name only, depending on how the file is executed). That pointer is then passed to 'strchr' to find the last backslash in the string.

However, the problem here is that this variable does not contain a pointer to a string at all: it is null. That is never supposed to occur, so it seems like certain parts of the C runtime in MSVCRT.DLL have not been initialized properly. What exactly went wrong here?

We decided to debug our TangleCrypt unpacked executable (but still packed with VMProtect), which does not crash. Putting a breakpoint-on-memory-access on the variable pointed to by the 'pgmptr' export (WinDbg: "ba r 8 msvcrt!pgmptr") caused the process to break at 0x401FF1, allowing us to confirm that the variable contained a pointer to an actual string, which proved that TangleCrypt was somehow responsible for the incomplete C runtime initialization.





Figure 26A/B: WinDbg disassembly of the same STONESTOP code in the unpacked executable. Register 'rax' is not 0 here, but is a pointer to the executable path.

Idea 1: TLS

One thing we noticed was that the unpacked executable has [Thread Local Storage \(TLS\) information](#) and a few TLS callbacks - functions typically used for data initialization/uninitialization. Here we have three such functions: 0x5AC152, 0x404450 and 0x404420.



Figure 27: TLS information in the unpacked executable (red). At offset 0x18 there is a pointer to the array of TLS callbacks (green)

The original executable - before it was packed with VMProtect - only had the two last callbacks, which are part of the C runtime initialization; the first callback 0x5AC152 was added by VMProtect and is responsible for decrypting the original PE sections so that the next callbacks can be executed (clever!).

However, the TangleCrypt loader completely ignores the original TLS information: breakpoints that we set on all 3 callbacks were not triggered when we expected them to trigger, or not triggered at all.

For a moment, we thought that we had identified the root cause of the crash, but that turned out not to be the case. Both VMProtect and C runtime are designed in such a way that they do not rely on TLS data and callbacks: if the TLS information is discarded somehow, both then just take different code paths to take care of things. This was easily confirmed by removing the pointer to the TLS information in the PE header of the unpacked executable; it was still working fine.

We can't say that it is a good idea from the TangleCrypt authors to discard the original TLS information - on the contrary, because some PE files may rely on it - but at least this flaw is not responsible for the crash in this sample. We reached a dead end here.

Idea 2: A deep dive into the C runtime

Eventually we decided to disassemble MSVCRT.DLL itself, to find out where and how the runtime initializes the variable where the 'pgmptr' export points to. It turned out that this is only done in two internal functions, and that these functions are only called by the MSVCRT.DLL exported function [_getmainargs](#).



Figure 28: Disassembly of one of the internal MSVCRT.DLL functions that set the 'pgmptr' variable, which is in the end just a pointer to GetModuleFileNameA output

In other words, this suggests that the C runtime assumes that '_getmainargs' is always called at least once before the runtime is "exposed" to the actual program at the 'main()' function. And that is a safe assumption: when we debug the unpacked executable and set a breakpoint on both

'_getmainargs' and 0x4013E3 (the call to 'main()', virtualized by VMProtect), the target would hit the '_getmainargs' breakpoint first, which proves that it is indeed called during the C runtime initialization.

When we let the debugger return from here, we end up in a small function that starts at 0x401160, part of the C runtime stub in the original executable.



Figure 29: Disassembly of function 'sub_401160'

Then the obvious question is: what's different in the TangleCrypt packed sample? It looks like '_getmainargs' is not executed at all?

Let's debug the executable and set the same breakpoints - '_getmainargs' and 0x4013E3 - but also at 0x401160, the function that is supposed to call '_getmainargs'. If all goes well, we first break at 0x401160. From here, we trace a few instructions to see where the call at 0x40119E leads to.



Figure 30: Disassembly of function 'sub_404BD0'

The instruction at 0x404BD0 jumps to the address stored in 0x40B620, which is part of the Import Address Table (IAT). But instead of jumping to '_getmainargs', we end up in code belonging to the TangleCrypt loader!



Figure 31: Disassembly of the '_getmainargs' re-implementation in TangleCrypt
So what is happening here?

When TangleCrypt decrypts and prepares the payload, it also needs to make sure that all imports in the IAT of the decrypted payload are resolved (a task otherwise done by the Windows PE loader). For most imports, TangleCrypt will simply write its original function address to the IAT. But for a few specific API functions, the authors have provided an own implementation of which the address is written to the IAT instead.



Figure 32: Part of the MSVCRT.DLL import table of a memory dump with initialized IAT. All entries

have an address in the range 0x7FFE78xxxxxx - which is where MSVCRT.DLL is loaded - except the two entries at 0x40B620 and 0x40B6B8 that refer to 0x7FF69Dxxxxxx, where the executable, i.e. TangleCrypt, is loaded.

Writing an own implementation of an API function is typically done to change its behaviour, for example to fix a compatibility issue (as Windows does for certain programs) or to prevent the API from doing something - we suspect that TangleCrypt redirects the 'msvcrt!exit' function for that reason. But a re-implementation introduces the risk of being incomplete, and that's exactly the problem with the '_getmainargs' implementation in the TangleCrypt loader: it does not initialize 'pgmptr', which causes a crash down the line. Or at least when the sample is executed without admin rights; STONESTOP only uses 'pgmptr' in the code to restart itself elevated, which explains why the crash does not occur if it's immediately executed as admin.

These bugs that we identified - the discarded TLS information and the incomplete re-implementations - were the main inspiration to name this packer TangleCrypt.

The other sample

The remaining question is why the other sample - 'b1.exe' - works as expected in all scenarios, even though the functionality is identical to '4cBh.exe'. Actually, there are two reasons:

- While the '4cBh.exe' payload is dynamically linked with MSVCRT.DLL, the 'b1.exe' payload is statically linked with the CRT and therefore has all runtime functions such as '_getmainargs' included. Since the import does not exist, the TangleCrypt loader does not replace it with its own incomplete implementation.
- As discussed before, the 'b1.exe' payload is configured to be injected in a child process. We have determined that the replaced imports depend on the injection method: for example, if the payload is injected in a child process, TangleCrypt will not replace the MSVCRT.DLL exports '_getmainargs' or 'exit', but it will redirect USER32.DLL export 'MessageBoxA'. In other words, if '4cBh.exe' had been configured to use the child process method, it would have worked as expected.

Conclusion

EDR killers are constantly evolving, and threat actors will keep finding new ways to hide their payloads through different techniques. TangleCrypt is one such example of an attempt to hide the payload through various mechanisms.

The packer shows several thoughtful design choices, from its multi-layer encryption, encoding and compression techniques, to its ability to switch execution techniques using a simple configuration string. For a threat actor deploying EDR killers to evade defense mechanisms, a loader with this level of flexibility can be highly valuable. Its payload-agnostic architecture also makes it suitable for hiding virtually any malware the threat actor decides to bundle with it. Inconsistencies found in its loader implementation can cause certain packed executables to behave unexpectedly, which

may explain why TangleCrypt is not widely observed in the wild. However, this also highlights varying capabilities of malware development: rapid development, limited testing and general carelessness can introduce bugs that ultimately reduce its effectiveness.

TangleCrypt is therefore a reminder that adversaries continually experiment, iterate, and repurpose tooling in their efforts to bypass detection. As EDR killers and their supporting infrastructure continue to evolve, defenders must anticipate not only sophisticated techniques, but also fast-moving, imperfect implementations that may appear in real attacks.

WithSecure products detect TangleCrypt packed samples as “Trojan:W32/TangleCrypt.A”.

The author of this blogpost would like to thank Amjad Alsharafi for his contributions to the investigation.

Appendix A: YARA rules

[YARA rule for TangleCrypt - GitHub](#)

Appendix B: Indicators of Compromise (IOCs)

[List of STONESTOP and ABYSSWORKER samples - GitHub](#)

References

- ABYSSWORKER / POORTRY / BURNTCIGAR & STONESTOP:
 - <https://www.elastic.co/security-labs/abyssworker>
 - <https://www.linkedin.com/pulse/attackers-leveraging-microsoft-teams-defaults-quick-assist-p1u5c/>
 - <https://news.sophos.com/en-us/2024/08/27/burnt-cigar-2/>
 - <https://news.sophos.com/en-us/2022/12/13/signed-driver-malware-moves-up-the-software-trust-chain/>
- HeartCrypt:
 - <https://unit42.paloaltonetworks.com/packer-as-a-service-heartcrypt-malware/>
- Others:
 - <https://news.sophos.com/en-us/2025/08/06/shared-secret-edr-killer-in-the-kill-chain/>
 - <https://unit42.paloaltonetworks.com/medusa-ransomware-escalation-new-leak-site/>

Share

