Crypted Hearts: Exposing the HeartCrypt Packer-as-a-Service Operation

By Jerome Tujague, Daniel Bunce

Published: 2024-12-13 · Archived: 2026-04-29 02:14:00 UTC

Executive Summary

This article analyzes a new packer-as-a-service (PaaS) called HeartCrypt, which is used to protect malware. It has been in development since July 2023 and began sales in February 2024. We have identified examples of malware samples created by this service based on strings found in several development samples the operators used to test their work.

The operator of this service has advertised it through underground forums and Telegram. Its operators charge $20 per file to pack, supporting both Windows x86 and .NET payloads.

The majority of HeartCrypt customers are malware operators using families such as LummaStealer, Remcos and Rhadamanthys. However, we've also observed payloads from a wide variety of other crimeware families.

HeartCrypt packs malicious code into otherwise legitimate binaries. We have discovered binaries packed with HeartCrypt from both external and internal telemetry.

We have successfully extracted malicious code for payloads from thousands of HeartCrypt samples. A majority of the unpacked payloads contain configuration data, which we have used to cluster samples and identify malicious campaigns targeting various industries and regions.

Palo Alto Networks customers are better protected from the threats discussed in this article through the following products and services:

- Cortex XDR and XSIAM
- Advanced WildFire

If you think you might have been compromised or have an urgent matter, contact the Unit 42 Incident Response team.

Related Unit 42 Topics	LummaStealer, Quasar RAT, RedLine Stealer, Remcos RAT, Cybercrime

HeartCrypt Background

HeartCrypt was originally discovered through underground forums and reported by security researchers in February and March 2024. During HeartCrypt's eight months of operation, it has been used to pack over 2,000 malicious payloads, involving roughly 45 different malware families.

We found HeartCrypt used in recent LummaStealer campaigns, including one impersonating legitimate software vendors and another using fake CAPTCHAs. We have also observed cybercrime activity targeting Latin American countries, with Remcos and XWorm using the HeartCrypt packer.

We first observed HeartCrypt during routine investigations in late June 2024 and initially categorized it as an unnamed, custom packer. Over the next several weeks, we continued to find more malware families using this packer and decided to investigate further.

Using unique byte patterns found within the packed samples, we created hunting rules and identified thousands of samples dating back to mid-2023. After implementing processes to parse these samples at scale, we made several notable discoveries.

Our first discovery was that development appears to have begun in July 2023, with the PaaS launching around Feb. 17, 2024. Nearly 1,000 samples from this period contained either no payload or a test payload.

Second, the packed payload was consistently added as a resource to a legitimate binary, often with a random name, though early versions sometimes used names containing HeartCrypt. This led us to our third discovery, the identification of the packer's distributor.

The distributor of HeartCrypt marketed the PaaS across multiple platforms, including:

- Telegram
- BlackHatForums
- XSS.is
- Exploit.in

Advertisements state HeartCrypt supports 32-bit Windows payloads at $20 per crypt. Figure 1 shows an ad in a Telegram post and Figure 2 shows an ad in an XSS.is post.

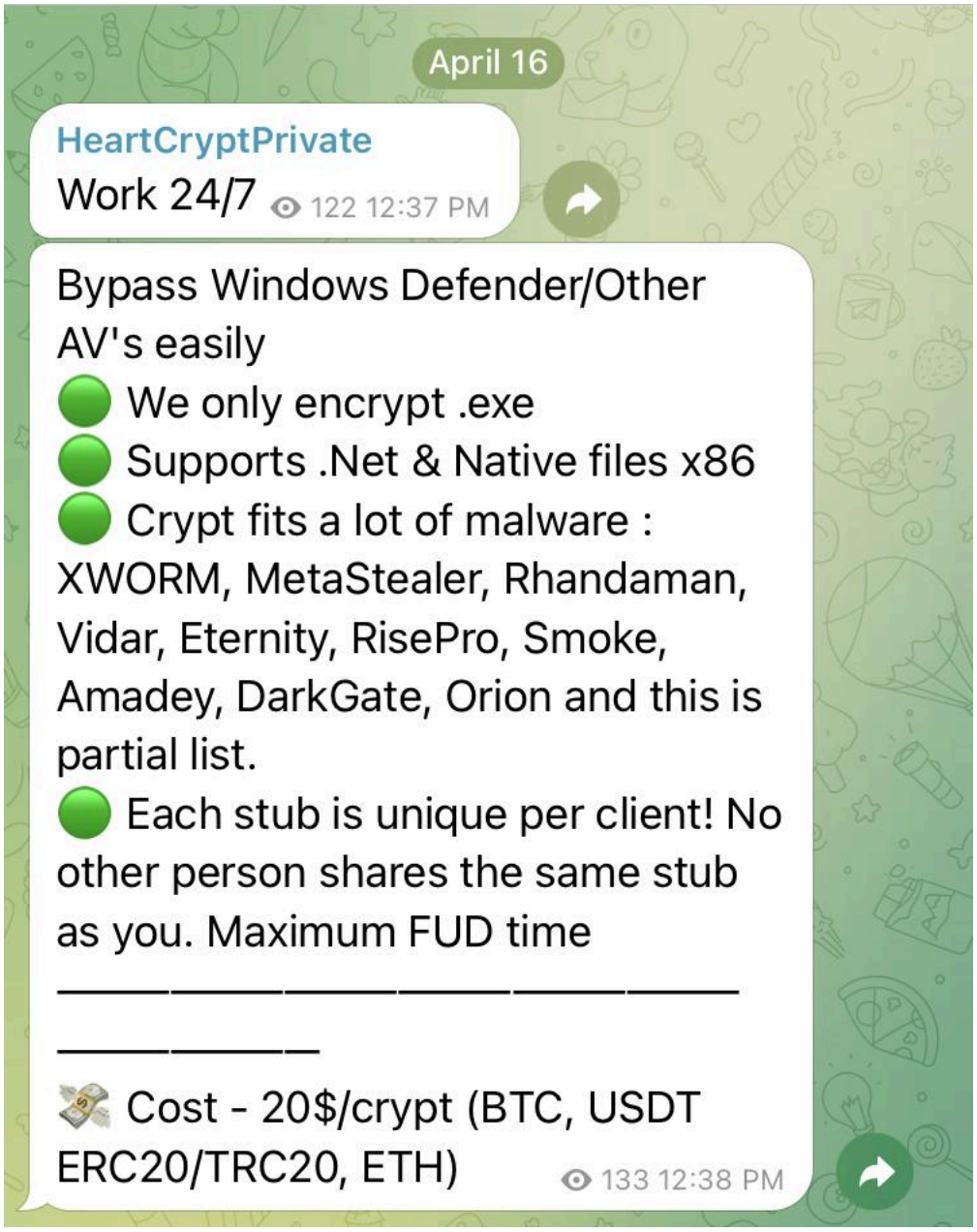

Figure 1. Post in the HeartCrypt Telegram channel.

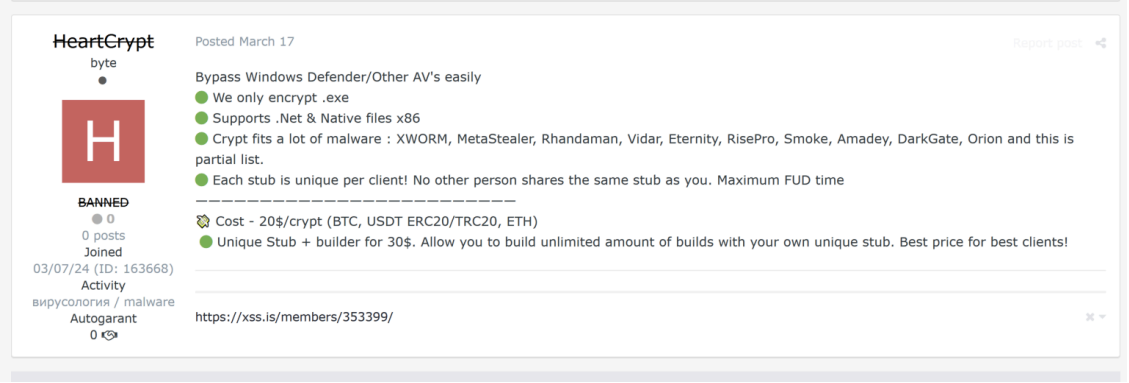

Figure 2. HeartCrypt XSS advertisement.

In HeartCrypt's PaaS model, customers submit their malware via Telegram or other private messaging services, where the operator then packs and returns it as a new binary. As we detail in the next section, the packing process

exemplifies how when even basic techniques are combined, it can create a challenge for reverse engineers.

HeartCrypt Technical Analysis

Creating the HeartCrypt Stub

The packing process begins by injecting malicious code into an otherwise legitimate executable file. This does not appear to be a random process. Our analysis reveals client-side customization.

We've identified over 300 distinct legitimate binaries its operators used as carriers for the malicious payload, which suggests a degree of client-side control. We theorize that the HeartCrypt service allows clients to select a specific binary for injection, tailoring the final payload to its intended target.

For example, a threat actor running a malware campaign based on a lure including an installer for a legitimate Windows application could request injection into a genuine but outdated installer. Distributed through a site impersonating the software vendor, the resulting packed malware would appear far more legitimate to a less-technical user, potentially increasing the likelihood of successful detonation.

The modification of the legitimate binary occurs in three key steps:

1. A contiguous block of code is added to the binary's .text section.
2. The control flow within the original binary is hijacked.
3. Several resources are added to the binary.

First, HeartCrypt adds a contiguous block of code to the binary's .text section. This code block is designed as position-independent code (PIC), a programming construct where the code's location in memory doesn't affect its execution. This allows the malicious code to run regardless of where it is loaded into memory by the operating system.

Secondly, HeartCrypt hijacks the control flow within the original binary. This is most often achieved by altering the start() function, the entry point for many executables. The modification typically involves adding a call or jmp instruction which redirects execution to the newly added PIC. Figure 3 shows a section of disassembled code from a HeartCrypt sample with an example of an added jmp instruction.

```
                public start
start           proc near

var_2C          = dword ptr -2Ch
var_28          = dword ptr -28h
var_20          = dword ptr -20h
ms_exc          = CPPEH_RECORD ptr -18h

; FUNCTION CHUNK AT .text:007C41E6 SIZE 00000056 BYTES

                push    ebp
                mov     ebp, esp
                push    0FFFFFFFFh
                push    offset stru_8B80A0
                push    offset sub_7CD5E4
                mov     eax, large fs:0
                push    eax
                mov     large fs:0, esp
                add     esp, 0FFFFFFE0h
                push    ebx
                push    esi
                push    edi
                mov     [ebp+ms_exc.old_esp], esp
                mov     eax, 94h
                push    offset sub_6C226B
                pop     eax
                add     eax, 1
                jmp     eax
start           endp ; sp-analysis failed
```

Added `jmp` function

Figure 3. HeartCrypt start() function modification.

The injected PIC leverages multiple control flow obfuscation methods to hinder analysis. These include:

- Stack strings
- Dynamic API resolution
- Hundreds of direct jmp instructions
- Non-returning functions
- Arithmetic operations that have no effect on program execution
- Junk bytes after jmp and call instructions, impeding disassembly and decompilation

The combination of these techniques makes both static and dynamic analysis extremely tedious.

With some effort, our analysis revealed that the initial PIC consists of two layers: an encoded block wrapped with a small decryption routine. The first layer uses specific byte patterns to identify the start and end of the encoded block. Figures 4 and 5 below show this as disassembled code from IDA Pro.

```
mov     byte ptr [ebp-0B78h], 52h ; 'R'
mov     byte ptr [ebp-0B77h], 0B9h
mov     byte ptr [ebp-0B76h], 10h
mov     byte ptr [ebp-0B75h], 10h
mov     byte ptr [ebp-0B74h], 10h
mov     byte ptr [ebp-0B73h], 10h
mov     byte ptr [ebp-0B72h], 5Ah ; 'Z'
```

Figure 4. Byte pattern built on the stack, indicating the start of the encrypted block.

```
mov     byte ptr [ebp-0B80h], 52h ; 'R'
mov     byte ptr [ebp-0B7Fh], 0B9h
mov     byte ptr [ebp-0B7Eh], 20h ; ' '
mov     byte ptr [ebp-0B7Dh], 20h ; ' '
mov     byte ptr [ebp-0B7Ch], 20h ; ' '
mov     byte ptr [ebp-0B7Bh], 20h ; ' '
mov     byte ptr [ebp-0B7Ah], 5Ah ; 'Z'
```

Figure 5. Byte pattern built on the stack, indicating the end of the encrypted block.

After locating the encoded block, the PIC performs a substitution operation on each byte, and execution passes directly to the decrypted block. The value used for substitution is chosen at random and is always in the range of one to nine.

The decrypted block uses the same obfuscation techniques as the first layer, again rendering static analysis infeasible. This second layer of the PIC iterates through the resources added to the binary and executes code within each in turn. Each iteration is performed in three steps.

The PIC first creates a stack string containing the resource name. Next, it leverages the FindResourceW, LoadResource and LockResource Windows APIs to acquire a pointer to the corresponding resource.

Finally, it uses VirtualProtect to modify the resource's memory protection attributes, enabling code execution. Execution is transferred directly to the resource, and upon completion, control is returned to the original PIC that restores the resource’s original memory protection using VirtualProtect. Figure 6 below provides a visual outline of the execution flow thus far.

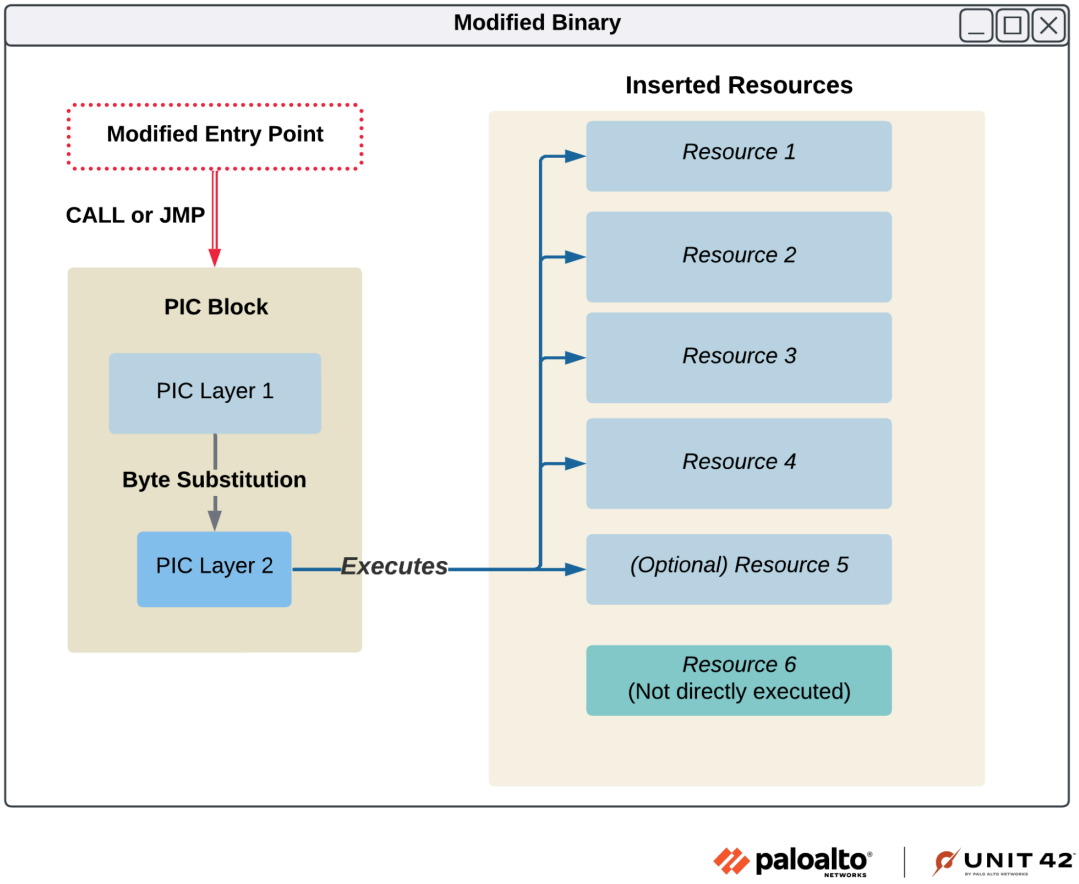

Figure 6. HeartCrypt's injected PIC executing code within each resource.

After hijacking the control flow, HeartCrypt adds several resources to the binary, each playing a key role in the packer's functionality and employing similar obfuscation across each layer. We now analyze each resource in detail, uncovering their individual functionalities and their collective contribution to the functionality of the packer.

Unraveling the Shellcode Resources

Each resource embedded in the binary contains PIC disguised as a bitmap (BMP) image file. This begins with a standard BMP header followed by a repeating hexadecimal pattern for padding.

Figure 7 shows an example of a resource PIC in a hex editor where you can see the first 2 bytes as the ASCII characters BM and the repeating hexadecimal pattern as 0x09.

```
00000000: 424d b8ad 0000 0000 0000 3600 0000 2800  BM........6...(.
00000010: 0000 b918 0000 0200 0000 0100 3400 0000  ............4...
00000020: 0000 5eb1 0000 0000 0000 0000 0000 0000  ..^.............
00000030: 0000 0000 0000 0909 0909 0909 0909 0909  ................
00000040: 0909 0909 0909 0909 0909 0909 0909 0909  ................
00000050: 0909 0909 0909 0909 0909 0909 0909 0909  ................
00000060: 0909 0909 0909 0909 0909 0909 0909 0909  ................
00000070: 0909 0909 0909 0909 0909 0909 0909 0909  ................
00000080: 0909 0909 0909 0909 0909 0909 0909 0909  ................
00000090: 0909 0909 0909 0909 0909 0909 0909 0909  ................
000000a0: 0909 0909 0909 0909 0909 0909 0909 0909  ................
000000b0: 0909 0909 0909 0909 0909 0909 0909 0909  ................
```

Figure 7. HeartCrypt resource PIC using a BMP header and padding bytes.

After the repeating hexadecimal pattern, the resource marks the start of its PIC with a sequence of bytes directly before the PIC's entry point. After identifying this sequence of bytes, the primary PIC transfers execution to the resource PIC.

Figure 8 shows this sequence of bytes later in the resource PIC as 0x13371337, just before the entry point.

```
000018b0: 0909 0909 0909 0909 0909 0909 0909 0909  ................
000018c0: 0909 0909 0909 0909 0909 0909 dede eeee  ................
000018d0: dede de13 3713 3755 8bec 81ec 1833 0000  ....7.7U.....3..
000018e0: b875 0000 0066 8985 14dc ffff b973 0000  .u...f.......s..
000018f0: 0066 898d 16dc ffff ba65 0000 0066 8995  .f.......e...f..
00001900: 18dc ffff b872 0000 0066 8985 1adc ffff  .....r...f......
00001910: b933 0000 0066 898d 1cdc ffff ba32 0000  .3...f.......2..
00001920: 0066 8995 1edc ffff b82e 0000 0066 8985  .f...........f..
00001930: 20dc ffff b964 0000 0066 898d 22dc ffff   ....d...f.."...
00001940: ba6c 0000 0066 8995 24dc ffff b86c 0000  .l...f..$....l..
00001950: 0066 8985 26dc ffff 33c9 6689 8d28 dcff  .f..&...3.f..(..
00001960: ffc6 8544 f2ff ff4d c685 45f2 ffff 65c6  ...D...M..E...e.
00001970: 8546 f2ff ff73 c685 47f2 ffff 73c6 8548  .F...s..G...s..H
```

Figure 8. Start of PIC in resource.

The resource PIC mirrors the structure of the initial PIC block in the legitimate binary, consisting of two layers with the same obfuscation techniques discussed previously. Each resource performs a different core function, with all observed HeartCrypt samples following the same pattern.

Resource 1: Anti-Dependency Emulation

The first resource appears designed to detect dependency emulation within a sandbox environment. It purposefully attempts to load non-existent DLLs via LoadLibraryW, specifically k7rn7l32.dll and ntd3ll.dll.

If the sandbox responds by generating a dummy DLL to prevent the program from crashing, HeartCrypt will call ExitProcess and terminate the execution. This is a rudimentary and unreliable method of sandbox detection, as modern sandboxes will typically return a controlled error code rather than creating a fake DLL. Further evidence of this functionality appeared in early development samples, where the author paired the stack-string CheckLibraryEmulated with MessageBoxW, likely for testing purposes.

Resource 2: Sandbox Loop Emulation Check

Earlier versions of the second resource (as with many of the other resources), provided useful insight into the functionality through debug strings. In this resource, the string CheckLoopEmulated, as well as the lack of timing-related API, allowed us to quickly identify what this resource could be responsible for.

The resource enters a while loop that performs a large number of mathematical calculations on an initial hard-coded value, similar to a hashing algorithm. The resulting hash is checked against an expected value.

If the two values match, the sample will set a flag value within memory to indicate the loop was not emulated or modified in any way. If this flag is not set, the process will call ExitProcess.

Resource 3: Windows Defender Evasion

The third resource provides anti-sandbox capabilities for evading Windows Defender. It leverages virtual DLLs (VDLLs), which are specialized versions of Windows DLLs within Defender's emulator, as described by Alexei Bulazel at BlackHat 2018 [PDF].

For example, within the emulator kernel32.dll has additional APIs such as MpReportEvent and MpAddToScanQueue. If HeartCrypt can load this API from kernel32, it can assume the sample is running within the Defender emulator.

This anti-sandbox technique was first reported in early April 2024 by Harfang Lab, in RaspberryRobin malware. It was adopted by the authors of HeartCrypt in the third resource just 15 days later.

Before adopting the Defender evasion technique, HeartCrypt included a different anti-sandbox technique that attempted to load d3d9::Direct3DCreate9. From our analysis, we believe this lines up with an anti-sandbox/anti-VM technique found within the InviZzzible virtual environment assessor, developed by Check Point Research.

The technique involves using the GetAdapterIdentifier function within an IDirect3D9 object to see if the vendor ID aligns with known VM providers. Alternatively, HeartCrypt's authors could also have implemented this technique under the assumption that a sandbox would be unlikely to provide Direct3D functionality. For example, if the sample failed to load the d3d9 library, it would terminate.

Resource 4: Final Payload Execution

The fourth resource decrypts and injects the final payload by accessing another embedded resource that holds the encoded payload. This resource masquerades as a BMP file but does not have the additional padding bytes or PIC. Instead, the BMP header is simply appended to the encoded payload.

The payload is a Windows executable binary encoded via a single-byte XOR operation rotating over a key hard-coded in the resource PIC as a stack string. We've identified over 50 distinct XOR keys across all HeartCrypt samples, with no discernable pattern. It is possible that the customer provides the key, but at this time we have no way to validate this theory.

After decryption, the PIC parses the decoded PE header to determine if the final payload is a .NET assembly or a natively compiled executable. If the packed sample is .NET, HeartCrypt will attempt to launch csc.exe (or in some

cases AppLaunch.exe) from the Microsoft .NET Framework directory. It then performs process hollowing on the spawned process, injecting and executing the final payload within it.

If the sample is not a .NET assembly, HeartCrypt spawns a copy of itself and injects the final payload using a similar process hollowing technique. While process hollowing is the primary method of injection, we have identified a sample that references NtQueueApcThread, suggesting that the developer has invested effort into diversifying the injection methods.

Resource 5: HeartCrypt Persistence

The fifth resource appears to be optional, as it isn't present in every sample we've identified. Its purpose is to establish persistence on the system using the HKCU\Software\Microsoft\Windows\CurrentVersion\Run registry key.

HeartCrypt drops an inflated version of itself onto the file system, adding several hundred thousand kilobytes of null padding before saving it to a hard-coded file path. It then sets the CurrentVersion\Run key to point to this file. To modify the registry, HeartCrypt uses either Windows API functions or the reg add command via cmd.exe.

Figure 9 below provides a visual representation of the HeartCrypt execution flow in its entirety.

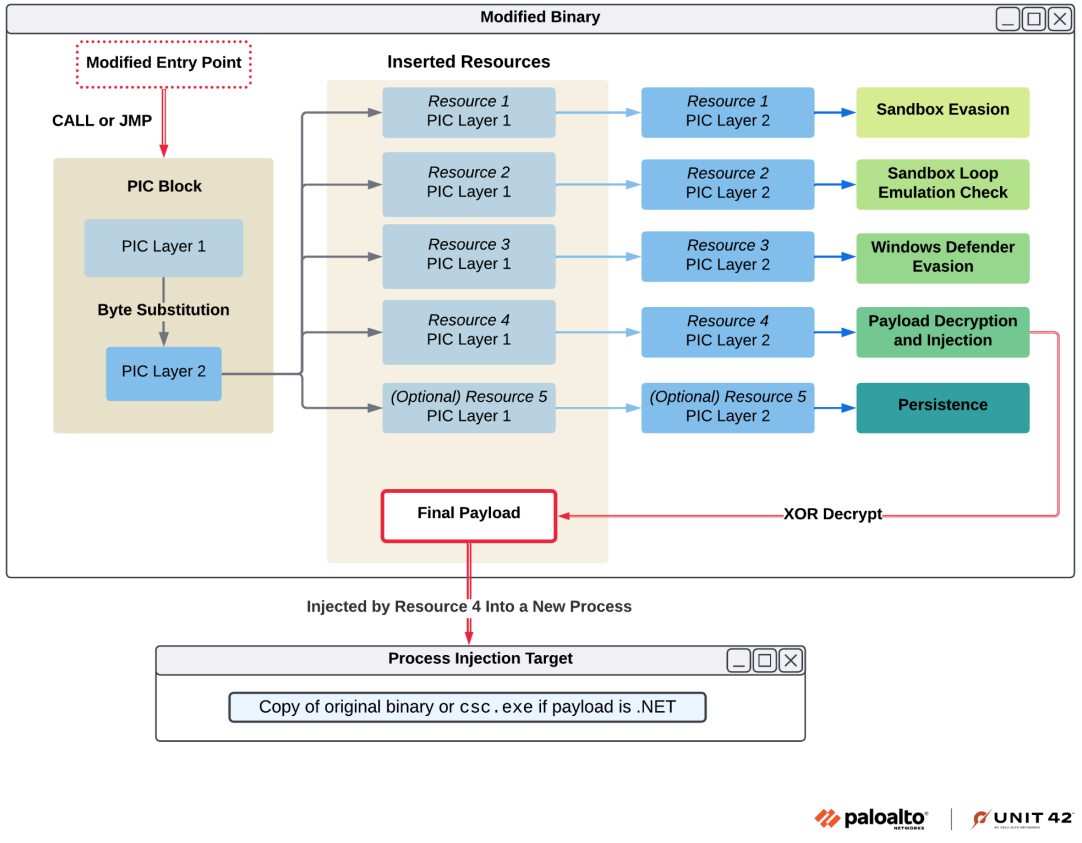

Figure 9. HeartCrypt execution flow.

Extracting HeartCrypt Payloads

Having detailed the individual functions of each embedded resource within the HeartCrypt packer, our next step was to automate the process of extracting payloads for further analysis. This involved developing a script capable of identifying the XOR key within the BMP-disguised resources.

Although HeartCrypt's obfuscation greatly hinders static analysis efforts, extracting key information is relatively trivial. The encoded payload is always a single-byte XORed Windows binary, so we can use a few basic methods to brute-force the key.

The first step is to locate the start of the encoded payload within the resource, which is always at the same offset. We can assume the first 2 bytes of the encoded payload will decode to MZ (0x4D5A), the Windows PE magic bytes found at the start of all executable files. As XOR operations are reversible, we can XOR the encoded bytes with 0x4D5A, resulting in the first 2 bytes of the XOR key.

Unencoded Windows executable files always contain multiple blocks of null bytes—for example, right after the section headers and just before the .text section. When a null byte is used in a single-byte XOR operation, the result is the byte used to perform the XOR. Therefore, we know that when the payload is encoded, the XOR key will be exposed in these blocks.

Once we've identified the initial bytes of the XOR key, we can search the entire binary for sequences beginning with these 2 bytes, resulting in a list of possible keys. We then attempt to decode the payload using each possible key, and if the resulting data is a valid PE file, we can assume we've identified the correct key.

While the brute-force method worked successfully for every sample of HeartCrypt we encountered, we updated our method to take a more efficient approach.

As we discussed earlier, each HeartCrypt resource includes a PIC block structured in two layers: the first layer applies a single-byte substitution operation to decode the second. By using frequency analysis, we can quickly identify the substitution key.

In our manual analysis of a decoded second-layer HeartCrypt resource PIC, we observed that the bytes 0x00 and 0xFF appeared most frequently. We know the encoding process involves adding a fixed value to each byte. Given that 0x00 is the most common value in the decoded PIC, the most common byte in the encoded PIC will indicate the substitution key. We implemented this logic into our script, and it was successful in decoding the first two layers of PIC resource blocks in all HeartCrypt samples.

The fourth HeartCrypt resource contains the XOR key stored as a stack string in the second layer PIC. Once we automated the process of decoding the PIC, we implemented a simple regex to extract all stack strings, allowing us to identify the XOR key for each sample without relying on brute force.

Ultimately, we were able to extract final payloads from all samples of HeartCrypt and perform further processing such as configuration extraction, when applicable.

Malicious Campaigns Using HeartCrypt

Analyzing the data gathered from our internal telemetry, we were able to get a better understanding of HeartCrypt activity. Our analysis shows there are just under 10 new samples of HeartCrypt found on average each day, with

occasional peaks of 60 samples as shown in Figure 10.

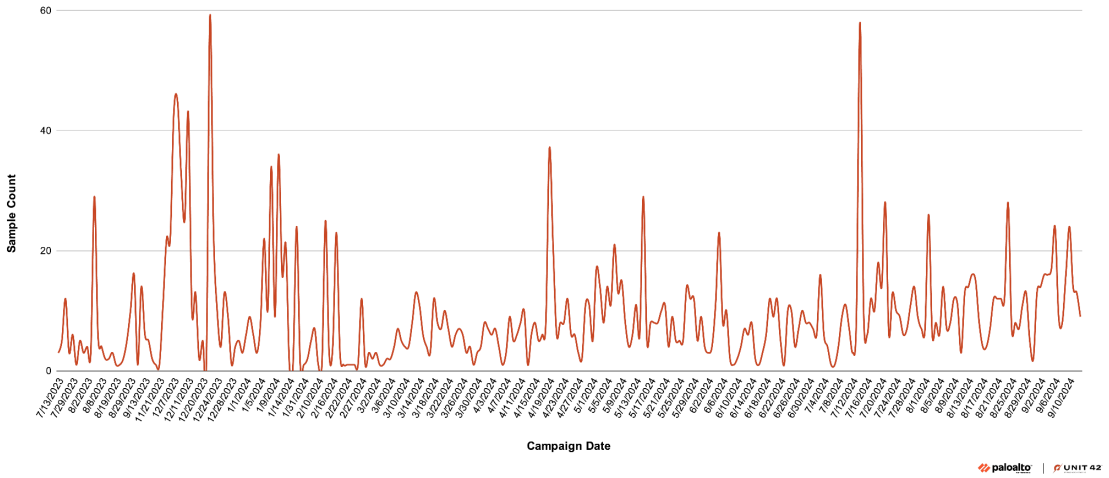

Figure 10. HeartCrypt samples identified over time from our internal telemetry.

Some of these peaks occurred during the developmental phase, between June 2023 and mid-February 2024. These samples had no payloads or test payloads using 127.0.0.1 as the C2 address, and many contained debug strings within PIC layers.

Our analysis indicates that the payload XOR keys appear to have some level of client-side customization. Across all samples, we found approximately 55 XOR keys consisting of distinct ASCII strings with different themes. These themes include months indicating the campaign, EDR/AV software company names, as well as random strings as shown in Figure 11 below.

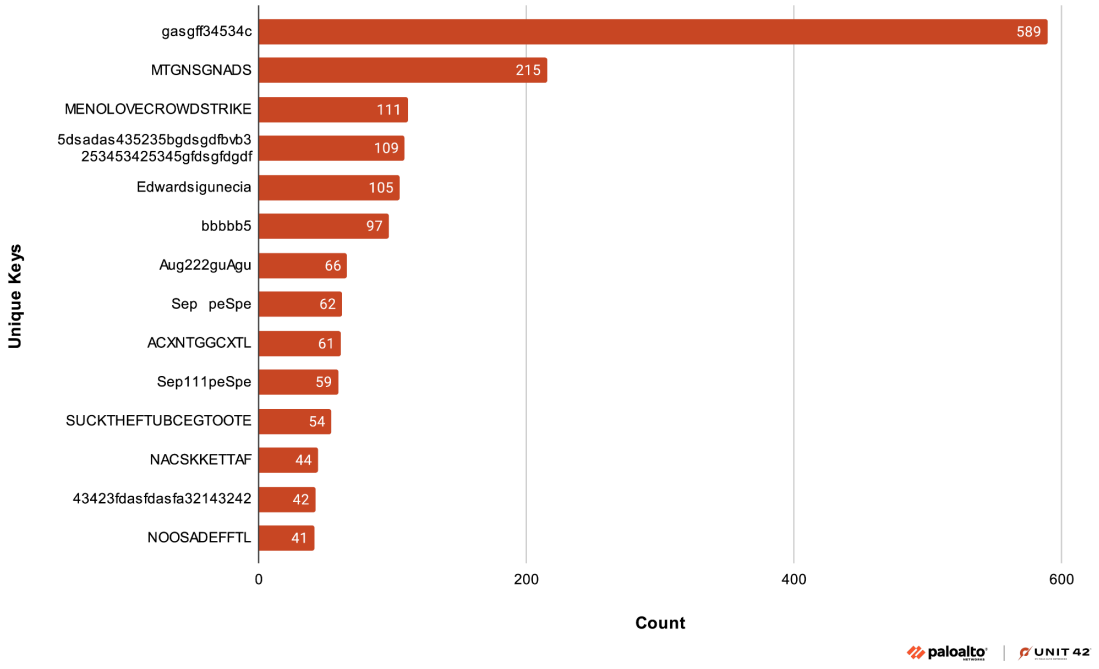

Figure 11. HeartCrypt XOR key usage across identified samples.

Automatic extraction of the payloads allowed us to cluster samples according to the identified malware family, as shown below in Figure 12.

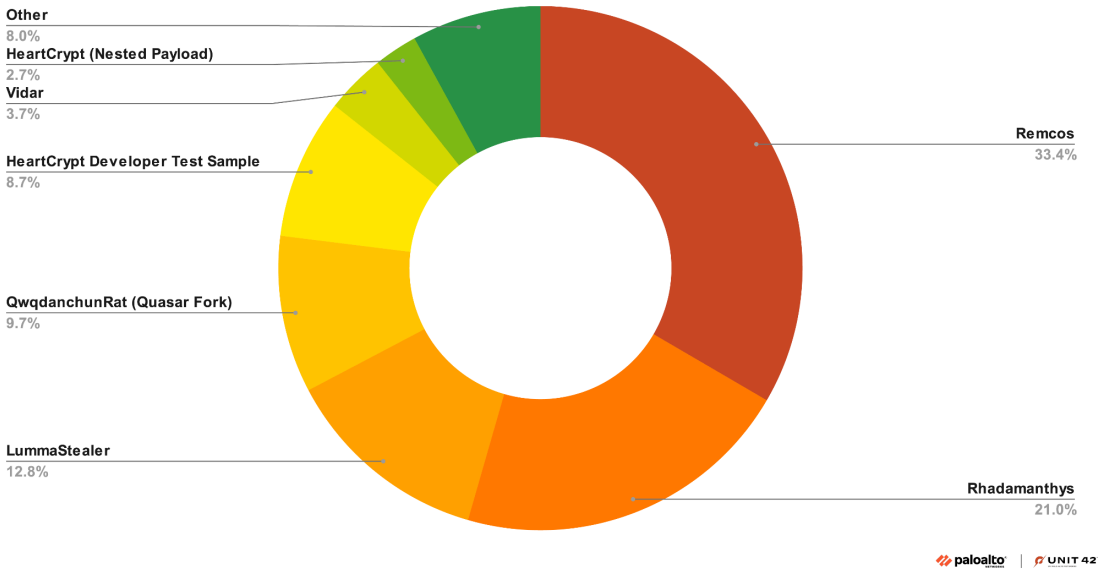

Figure 12. Malware families extracted from HeartCrypt samples.

Remcos is the payload most frequently seen across all HeartCrypt samples, because HeartCrypt's developers often used it during their development cycle as a test payload. We have also observed several clusters of Remcos targeting Latin American countries in recent months. For further details, see the Indicators of Compromise section of this article.

Lumma Steal is another payload frequently deployed by HeartCrypt packed samples. We have recently identified HeartCrypt samples from a previously reported LummaStealer campaign impersonating software vendors we originally posted about in October 2024.

We have also discovered HeartCrypt packed LummaStealer samples from a campaign using fake CAPTCHAs and copy/paste PowerShell script similar to one we originally reported on in August 2024. These campaigns have remained active since then.

Conclusion

Our analysis of HeartCrypt – a PaaS actively used by various threat actors – reveals what its samples look like in the wild, including extracting payloads for grouping. We documented HeartCrypt's evolution from its initial development in July 2023 to its February 2024 launch, tracking its use in over 2,000 malicious payloads across 45 malware families.

The packer's obfuscation techniques combine PIC, multiple layers of encoding and resource-based execution to significantly hinder analysis. Marketed on various underground forums, HeartCrypt's PaaS model lowers the barrier to entry for malware operators, increasing the volume and success of infections.

This lowered barrier to entry highlights the need for defenders to practice proactive threat hunting, focusing on identifying unique byte patterns and packer characteristics to detect obfuscated malware. Furthermore, the ease with which threat actors can leverage services like HeartCrypt showcases the continuous commoditization of malware development.

Palo Alto Networks customers are better protected from the threats discussed in this article through the following products and services:

- Cortex XDR and XSIAM
- Advanced WildFire

If you think you may have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America Toll-Free: 866.486.4842 (866.4.UNIT42)
- EMEA: +31.20.299.3130
- APAC: +65.6983.8730
- Japan: +81.50.1790.0200

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

HeartCrypt YARA Rule

```
1  <span style="font-weight: 400;">rule u42_crime_win_heartcrypt</span>
2  {
3  meta:
4  author = "Unit 42 Threat Intelligence"
5  date = "2024-11-30"
6  description = "HeartCrypt PaaS hunting rule."
7  hash = "7f4d6a371e872d8b4999d415401589c32adcfc6cfc26892cfa3316e4fccec270"
8  strings:
9  $a = {E8 08 00 00 00 00 ?? ?? ?? ?? ?? ?? ?? 83 C4 04 81}
10 $b = {
11 B8 4D 00 00 00
12 66 89 85 ?? ?? ?? ??
13 B9 42 00 00 00
14 66 89 8D ?? ?? ?? ??
```

```
15  BA 53 00 00 00
16  66 89 95 ?? ?? ?? ??
17  B8 65 00 00 00
18  66 89 85 ?? ?? ?? ??
19  B9 72 00 00 00
20  66 89 8D ?? ?? ?? ??
21  BA 76 00 00 00
22  66 89 95 ?? ?? ?? ??
23  B8 69 00 00 00
24  66 89 85 ?? ?? ?? ??
25  B9 63 00 00 00
26  66 89 8D ?? ?? ?? ??
27  BA 65 00 00 00
28  66 89 95 ?? ?? ?? ??
29  B8 2E 00 00 00
30  66 89 85 ?? ?? ?? ??
31  B9 65 00 00 00
32  66 89 8D ?? ?? ?? ??
33  BA 78 00 00 00
34  66 89 95 ?? ?? ?? ??
35  B8 65 00 00 00
36  66 89 85 ?? ?? ?? ??
37  33 C9
38  66 89 8D ?? ?? ?? ??
39  }
40  condition:
```

```
41  $a or $b
42  }
```

Indicators of Compromise

A text-based CSV spreadsheet for the HeartCrypt samples we have identified so far is available at a link from our GitHub repository.
