Buer Loader Analysis, a Rusted malware program - TEHTRIS

By Laurent Oudot

Published: 2022-01-20 · Archived: 2026-04-05 19:13:32 UTC

Malware analysis is part of the CTI team's daily routine. This article presents the analysis of a Rust strain of Buer Loader from the reception of the samples to the writing of a stage2* extraction script. Despite several protection mechanisms, it was possible to extract all the samples in different ways. TEHTRIS provides the code for such an extraction.

Introduction

The Rust language [1] is more and more used [2] by developers. Indeed, the philosophy of this language is to maximize security while offering performances close to C++ in a pleasant syntax. Many ambitious projects like Kerla [3] aim at offering an operating system that is compatible with any Linux binary, without modification. This gives Rust a lot of credibility and explains its adoption by developers. From a low-level point of view, this language generates a more complex binary code to sign and analyze compared to C by offering more instructions and adding obfuscation at low cost, which is of interest to malware authors.

If still few adopt it, it is however not surprising to note that this language is starting to make a name for itself in the malware bestiary, as shown by Buer Loader [3] which includes variants developed in this language.

If this malware program has already been analyzed [3], it remains interesting to study its "stage1*" to identify the protection mechanisms and the evolutions concerning its obfuscation over time. Indeed, sandbox bypass techniques [5] and memory loading of offended binaries [6] by an unknown mechanism have been observed. This was enough to trigger the interest of a retro-analysis of the loader in question.

The goal is to characterize and extract the main load of the malware that was specifically spotted in computing environments between July and August 2021.

An emulation-based extraction method was presented at the Hack-It-N conference in December 2021. A replay of the presentation is available on Youtube [16].

The retrieved samples work on the same model, i.e. a base in Rust loading in memory a stage2, itself written in Rust. Here is the list of these samples:

Sha256	Compile time	Size

001405ded84e227092bafe165117888d423719d7d75554025ec410d1d6558925	2021-07-28 17:59:19	3.4 MB
4421dbc01ddc5ed959419fe2a3a0f1c7b48f92b880273b481eb249cd17d59b91	2021-08-11 15:35:55	6.6 MB
52d8316b0765c147558aecbda686d076783f3a08b2741b8c9e3e717cc56e8a92	2021-07-19 13:57:22	7.3 MB
580d55f1e51465b697d46e67561f3161d4534a73e8aa47e18b9bae344d46bcf4	2021-07-19 13:57:22	7.3 MB
578dc62dfa0203080da262676f28c679114d6b1c90a4ab6c07b736d9ce64e43e	2021-07-15 16:28:54	7.1 MB
5ac6766680c8c06a4b0b4e6a929ec4f5404fca75aa774f3eb986f81b1b30622b	2021-08-05 14:47:25	4.9 MB
64dd547546394e1d431a25a671892c7aca9cf57ed0733a7435028792ad42f4a7	2021-08-16 18:54:29	4.1 MB
88689636f4b2287701b63f42c12e7e2387bf4c3ecc45eeb8a61ea707126bad9b	2021-08-03 15:46:42	3.3 MB
afb5cbe324865253c7a9dcadbe66c66746ea360f0cd184a2f4e1bbf104533ccd	2021-06-25 17:49:48	7.1 MB
c425264f34fa8574c7e4321020eb374b9364a094cda9647e557b97d5e2b8c17b	2021-08-16 18:54:29	4.1 MB
d3a486d3b032834b1203adefd25d0bf0b36fae7f9e72071c21ccc266e1e1f893	2021-07-19 14:14:54	4.3 MB

The compilation dates seem consistent with each other. Indeed, they match approximately the submission dates on Virus Total [15] (with a small delay). The binaries are stripped (i.e. symbols unnecessary for the binary to work

have been removed, especially function names and debugging symbols) but still contain information about the used source code:

Unfortunately, BinDiff [7] is not able to compare binaries. The control flow is too complex for it. Therefore, the code comparisons were done manually by our experts.

BinDiff error

Our analysis determined that there seem to be only 2 source files outside of the open source (mainly cryptographic) libraries used. A first anti-sandbox technique consists in wasting time by performing an unnecessary loop. The order of magnitude of the time needed (and lost) is about one minute with a 100% occupied CPU.

Anti-sandbox by waste of time

After passing through this loop, we arrive at the decryption function of stage2. This one contains a large number of successive calls to useless functions, among which we find:

- **GetCommandLineA;**
- **GetCurrentProcess;**
- **GetEnvironmentStrings;**
- **GetLastError;**
- **GetProcessHeap;**
- **GetTickCount;**

It is probably a saturation mechanism using calls (call to a Windows library) and not taking any argument to override possible sandboxes, which complicates code emulation. It seems that a script generates all the calls. This technique also has the advantage of adding a credible call/instruction ratio compared to a legitimate program. Counting the calls sometimes allows to find decryption or unpacking routines:

Decoys variant 1

Decoys variant 2

These calls are repeated throughout the desobfuscation routine. The data are successively pushed on the stack, then in the heap with the help of kernel32 !HeapAlloc, in the form of DWORD:

Stage2 buffer reconstruction

The call graph is very different across samples, reinforcing our hypothesis that the code is generated from a script. The mix of calls and desobfuscation involves a special effort to blend in a behavior that appears to be legitimate.

Control flow Variant1

Control flow Variant2

Control flow Variant3

This technique allows to include binary data by limiting the entropy between 0.5 and 0.8, very close to what one would expect from legitimate instructions.

Entropy of the malware program

This data, once reconstituted, is placed in a buffer which is decrypted. We then recognize a Key Schedule type mechanism, followed by a generator reminiscent of RC4:

KSA block

RC4 Random generator

The Rust paradigm makes these decryption routines difficult to identify. Tools such as findcrypt [8] do not detect them. The following check identifies the entire desobfuscation chain:

Decryption in python

The decompression is performed by the version 0.1.3 of the lzma-rs library [9], and the decryption keys are easily visible in the code. The following script allows its extraction:

Key extraction script

The list of RC4 keys for the samples is as follows:

- **YDVHHCYTCH ;**
- **DQOQHMLGYU ;**
- **BWSCVQZXOB ;**
- **SIMHIDVSCR ;**

- **RGZPMAAQRP ;**
- **YVMOOVSIOF ;**
- **KSQKGUUTXZ ;**
- **EUWPUQYDTT ;**
- **KHBXNNHKNN ;**

Unfortunately, the number of keys is insufficient to evaluate the quality of the PRNG [10]. Note that the charset is very small, which does not matter since the key is in plain text in the ".rdata" section. Given the extremely variable and complex call graph, writing a generic data extraction script that works for all samples is tedious. Now that we know the key (cf. previous extraction script), we need to find the encrypted data. These, once desobfuscated, have a header with invariant data:

3 blocks of encrypted buffer header

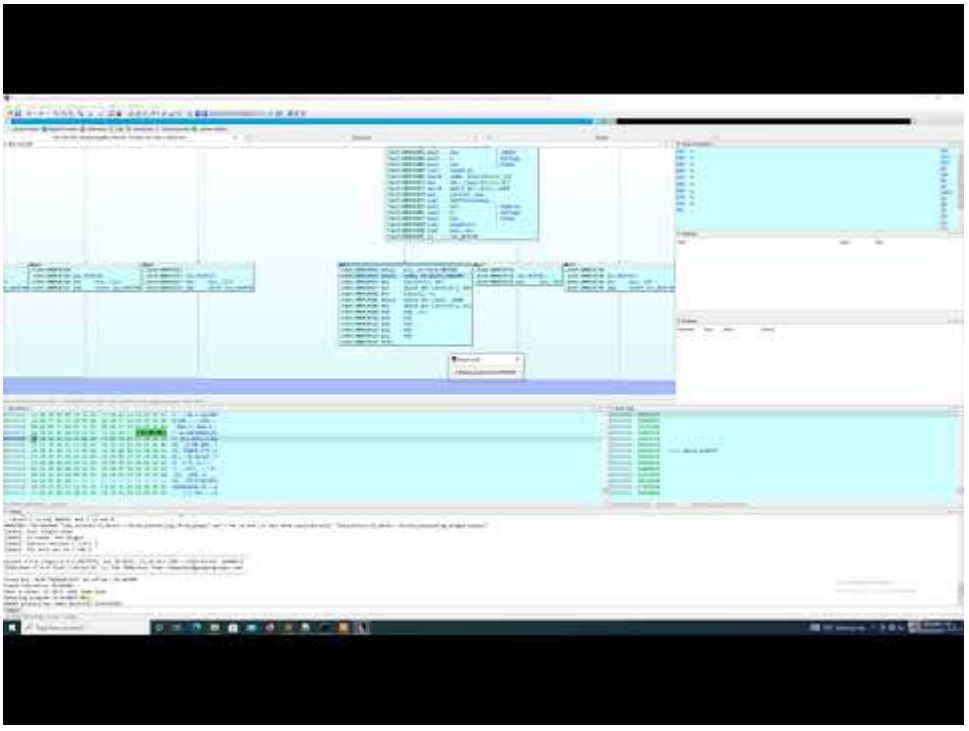
The following script then allows to find the extracted data in memory, provided that the program has performed the decryption phase by carrying out an Egg-hunting search:

Encrypted data recovery script

The only thing missing to automate the extraction is to find an address or to set a breakpoint, which is done in a fairly logical way by finding the references to the RC4 key:

Malware launching script

And the script works on the first try for all samples. The source code of the tool is available on the TEHTRIS github [19].



An alternative method was presented at the Hack-it-N conference [16]. It consists in extracting the stage2 by emulation using the unicorn lib [17]. The source code is available on the TEHTRIS github [18].

The extraction of stage2 from the studied samples gives the following list:

Sha256	**Stage1 compile time**	**Stage 2 compile time**
edc3b5f8d45d7a1cceee144e57fc5ddfaf8c0c7407a1514d2f3bab4f3c9f18b8	2021-07-28 17:59:19	2021-07-28 17:39:31
d7ec38c0e89a749a7727e5644328835b50e19302e9f3a4688809403ebcbd03d2	2021-08-11 15:35:55	2021-08-11 15:30:38
6578db32dc78ef7f41213557cf894d03b97ed6974ae7a72bec9b7c7ac08c4ba9	2021-07-19 13:57:22	2021-07-19 13:44:11
d48d91451b9594eadc0d1ef6e379bbce9a6033bd337e06d46613a70187c9c5ef	2021-07-15 16:28:54	2021-07-15 16:20:18

54109b12cbbd223f5ad79a9f87bfe50ef05a80e5551a3c1931748c3698900496	2021-08-05 14:47:25	2021-08-05 14:38:43
2d8a2bcc45daedd343eadb4222885d12a221bebbf7f1d98f92cb233df0a4c1d4	2021-08-16 18:54:29	2021-08-16 18:45:22
16feaed6222ce4a1941ae0c32eabaf0ecf68c33c49544f71d431d1b70c4247fd	2021-08-03 15:46:42	2021-08-03 15:38:18
7af554fb260817350d33b801d9f0b8a638b831992f4b1b31c2bbdab875b211df	2021-06-25 17:49:48	2021-06-25 17:43:23
2d8a2bcc45daedd343eadb4222885d12a221bebbf7f1d98f92cb233df0a4c1d4	2021-08-16 18:54:29	2021-08-16 18:45:22
039d63a07372e6e17f9779ccffbafbf9a06a9402ade58fbec3b0b2f8d2038175	2021-07-19 13:57:22	2021-07-19 13:46:46

The timestamps seem coherent between themselves and we note that the compilation dates correspond with a gap of 5 to 10 minutes, suggesting a manual packing step.

We note that stage2 verifies the presence of a virtual machine by testing the presence of the following executables:

> vboxservice.exe ;
> vboxtray.exe ;
> vmtoolsd.exe ;
> vmwaretray.exe ;
> vmwareuser.exe ;
> vmacthlp.exe ;
> vmsrvc.exe ;
> vmusrvc.exe ;
> prl_cc.exe ;
> prl_tools.exe ;
> xenservice.exe ;
> qemu-ga.exe ;
> windanr.exe.

Once the desobfuscation step is done, the binary is no longer obfuscated and easily delivers its secrets. For example C2 [11]:

List of C2 URLs

Or encrypts list of files constituting the source code:

Source code metadata

The analysis of the stage2 has already been done, however, and we will not describe it here.

Conclusion

Evading automated malware detection systems is a balancing act between binary entropy, hiding encryption functions, slowing down the analysis time, obtaining a consistent call/instruction ratio…

The use of Rust adds a machine code overlay which is however much less than what a Go [12], delphi [13], cython [14], etc. compiler would do, making a manual analysis quite reasonable.

However, these techniques are no match for a reverser or a well-configured sandbox. The manual analysis of the most known and least traceable threats is a plus in the continuous improvement of our sandbox and EDR products. It is very likely that in the future, more and more malware programs will be developed in Rust.

* The use of stageN consists in separating the malware program into several specialized sub-parts in order to escape antivirus detection. In this case, stage1 is the malware program as sent to victims and stage2 is the payload encapsulated in stage1.

BIBLIOGRAPHY

[1] https://www.rust-lang.org/

[2] https://www.tiobe.com/tiobe-index/rust/

[3] https://github.com/nuta/kerla

[4] https://www.proofpoint.com/us/blog/threat-insight/new-variant-buer-loader-written-rust

[5] https://fr.wikipedia.org/wiki/Sandbox_(s%C3%A9curit%C3%A9_informatique)

[6] https://fr.wikipedia.org/wiki/Offuscation

[7] https://www.zynamics.com/bindiff.html

[8] https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt

[9] https://github.com/gendx/lzma-rs

[10] https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_nombres_pseudo-al%C3%A9atoires

[11] https://www.trendmicro.com/vinfo/us/security/definition/command-and-control-server

[12] https://golang.org/

[13] https://www.embarcadero.com/fr/products/delphi

[14] https://cython.org/

[15] https://www.virustotal.com

[16] https://www.youtube.com/watch?v=4Lux_0IROMY

[17] https://www.unicorn-engine.org/

[18] https://github.com/tehtris-hub/MalwareTool/blob/main/Buer/extract_buer.py

[19] https://github.com/tehtris-hub/MalwareTool/blob/main/Buer/extract_buer_debug.py

---