



As already mentioned in [my last MuddyWater article](#), I originally planned to take a closer look at the remaining RustyWater samples in the hope of finding additional leftover artefacts. But: writing a pure “let’s grep a few more binaries for leftovers” follow-up article sounded painfully boring, both to read and to write. So instead of doing that, I decided to version the individual samples properly and build a small change-tracking timeline around them.

Mostly out of curiosity, partly out of stubbornness.

At least this way we get something that resembles actual analysis rather than archaeological string hunting.

Before diving into version diffs and timelines, it’s worth briefly grounding what we’re actually dealing with here, without turning this into a full-blown reverse engineering novella.

RustyStealer (sometimes referenced as RustyWater, Archer RAT or related variants in public reporting) is a Rust-based information stealer / RAT that has been observed in MuddyWater-attributed campaigns. In practice it behaves like a fairly typical post-compromise implant: basic host reconnaissance, security product awareness, persistence via registry, C2 communication and modular task execution. Nothing magical, nothing revolutionary, just a modernized tooling stack implemented in Rust instead of the usual C/C++ ecosystem. [Here’s an article from cloudseek.com](#) if you are interested in the details of it’s deliver and execution nature.

And that’s exactly where this article deliberately does *not* go: this is not a line-by-line teardown of every decoding routine, syscall wrapper or math loop. There are already reports on the internet that lovingly explain how a single function increments a counter.

What I’m interested in instead is how this thing evolves over time and how to track it

How do the binaries change between builds?

Which libs appear or disappear?

When does the architecture shift?

Where do experiments get rolled back?
Which parts get hardened, obfuscated or simplified?

The goal is to treat this malware family more like a software project with a messy commit history than a static artifact. By correlating compile timestamps, dependency fingerprints, fuzzy hashes, embedded artefacts and targeted code diffs, we can reconstruct a surprisingly coherent development timeline and extract signals about tooling maturity, experimentation phases and operational priorities .

And honestly: that's way more fun than grepping strings all day and more useful for my threat actor research

For the analysis I used the following samples:

Hash	Date
e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd	08.01.2026
ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914	08.01.2026
7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58	08.01.2026
e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108	08.01.2026
a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79	10.01.2026

My First Step: I always start with the most low-effort signal: raw strings.
Sometimes you get lucky and developers leave behind absolute paths, usernames, toolchain artefacts or other accidental breadcrumbs.

The first thing I wanted to verify was whether any additional user paths were embedded in the samples:

```
for x in *.exe; do strings "$x" | r="$(strings "$x" | grep 'C:'); if ! [ -z "$r" ]; then echo "$x $r\n"; fi ; done
```

```
000 0 08.01.2026 for x in *.exe; do strings "$x" | r="$(strings "$x" | grep 'C:'); if ! [ -z "$r" ]; then echo "$x $r\n"; fi ; done
7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58 .exe C:\D%
egistry\src\index.crates.io-1949cf8c6b5b557fddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914.exe C:\ProgrH
<;C:<;t

e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd.exe C:\ProgrH
<;C:<;t

e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe C:\ProgrH
<;C:<;t
C:\Users\Bruno\Desktop\prt.exe

000 0 08.01.2026 strings e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe | grep "C:"
C:\ProgrH
<;C:<;t
C:\Users\Bruno\Desktop\art.exe
000 0 08.01.2026
```

And yes, in two of the binaries we do indeed find another leftover artefact:

```
C:\Users\Bruno\Desktop\art.exe
C:\Users\Jacob\...
```

Just to be clear: the developer is almost certainly not named *Bruno/Jacob*.
This is simply the username of the build environment used at that time. Still, it gives us a nice anchor point when correlating samples later on.

Dependency extraction and library fingerprinting:

Next, I wanted to extract the dependency information from the binaries:

```
# get llvm strings
for x in *.exe; do strings "$x" | r="$(strings "$x" | grep 'llvm'); if ! [ -z "$r" ]; then echo "$x $r\n"; fi ; done
# get crates with version and write to file sample.exe.strings
for x in *.exe; do strings "$x" | grep "crates\.io-" | cut -d'\ ' -f5 | sort | uniq >> "$x.strings" ; done
```

This gives us a list of Rust crates and libraries that were present at build time:

<pre> 08.01.2026 > E ddc6e6c76ac325d89799a50aff011ec69ed3b5341740619bde395b8068220c ... 1 aead-0.5.2 2 aes-gcm-0.10.3 3 atomic-waker-1.1.2 4 base64-0.22.1 5 bytes-1.10.1 6 cipher-0.4.4 7 ctr-0.9.2 8 futures-channel-0.3.31 9 futures-core-0.3.31 10 futures-util-0.3.31 11 generic-array-0.14.7 12 h2-0.4.12 13 hashbrown-0.15.5 14 http-1.3.1 15 httparse-1.10.1 16 http-body-util-0.1.3 17 hyper-1.7.0 18 hyper-tls-0.6.0 </pre>	<pre> 08.01.2026 > E e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c ... 1 aead-0.5.2 2 aes-gcm-0.10.3 3 atomic-waker-1.1.2 4 base64-0.22.1 5 bytes-1.10.1 6 cipher-0.4.4 7 ctr-0.9.2 8 futures-channel-0.3.31 9 futures-core-0.3.31 10 futures-util-0.3.31 11 generic-array-0.14.7 12 h2-0.4.12 13 hashbrown-0.15.5 14 http-1.3.1 15 httparse-1.10.1 16 http-body-util-0.1.3 17 hyper-1.7.0 18 hyper-tls-0.6.0 </pre>	<pre> 10.01.2026 > E a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461efc9eab26bd9f410c ... 1 aead-0.5.2 2 aes-gcm-0.10.3 3 atomic-waker-1.1.2 4 base64-0.22.1 5 bytes-1.10.1 6 cipher-0.4.4 7 ctr-0.9.2 8 futures-channel-0.3.31 9 futures-core-0.3.31 10 futures-util-0.3.31 11 generic-array-0.14.7 12 h2-0.4.12 13 hashbrown-0.15.5 14 http-1.3.1 15 httparse-1.10.1 16 http-body-util-0.1.3 17 hyper-1.7.0 18 hyper-tls-0.6.0 </pre>
--	---	--

From this output I consolidated a full list of all dependencies observed across all samples.
It already allows a first sanity check: are all samples built from the same dependency set or do we see deviations?

In total, the consolidated list contains 50 libraries, however, each individual sample only exposes around 44-45 crates.
That already tells us that not all builds are identical.

Here is the full list of observed libraries:

Library	Version
aead	0.5.2
aes	0.8.4
aes-gcm	0.10.3
atomic-waker	1.1.2
base64	0.22.1
bytes	1.10.1
cipher	0.4.4
ctr	0.9.2
futures-channel	0.3.31
futures-core	0.3.31
futures-util	0.3.31
generic-array	0.14.7
h2	0.4.12
hashbrown	0.15.5
http	1.3.1
httparse	1.10.1
http-body-util	1.7.0
hyper	1.7.0
hyper-tls	0.6.0
hyper-utils	0.1.16
icu_collections	2.0.0
icu_normalizer	2.0.0
idna	1.1.10
indexmap	2.11.0
ipnet	2.11.0
iri-string	0.7.8
mio	1.0.4
native-tls	0.2.14
percent-encoding	2.3.2
rand	0.6.5
rand_core	0.4.2
rand_hc	0.1.0
rand_jitter	0.1.4
reqwest	0.12.23
schannel	0.1.27
slab	0.4.11
smallvec	1.15.1
socket2	0.6.0

tokio	1.47.1
tokio-native-tls	0.3.1
tokio-util	0.7.16
tower	0.5.2
universal-hash	0.5.1
url	2.5.7
utf8_iter	1.0.4
want	0.3.1
windows-registry	0.5.3
winreg	0.55.0
rustc-demangle	0.1.24
rustc-demangle	0.1.26

Since we now know that differences exist, I built a per-sample matrix to visualize which libraries appear in which binary:

Library	e081b...6abbd	e61b2...83108	a2001...10c79	ddc6e...20914	7523e...33f58
aead-0.5.2					
aes-0.8.4					
aes-gcm-0.10.3					
atomic-waker-1.1.2					
base64-0.22.1					
bytes-1.10.1					
cipher-0.4.4					
ctr-0.9.2					
futures-channel-0.3.31					
futures-core-0.3.31					
futures-util-0.3.31					
generic-array-0.14.7					
h2-0.4.12					
hashbrown-0.15.5					
http-1.3.1					
httparse-1.10.1					
http-body-util-0.1.3					
hyper-1.7.0					
hyper-tls-0.6.0					
hyper-util-0.1.16					
icu_collections-2.0.0					
icu_normalizer-2.0.0					
idna-1.1.0					
indexmap-2.11.0					
ipnet-2.11.0					
iri-string-0.7.8					
mio-1.0.4					
native-tls-0.2.14					
percent-encoding-2.3.2					
rand-0.6.5					
rand_core-0.4.2					
rand_hc-0.1.0					
rand_jitter-0.1.4					
reqwest-0.12.23					
schannel-0.1.27					
slab-0.4.11					
smallvec-1.15.1					
socket2-0.6.0					
tokio-1.47.1					
tokio-native-tls-0.3.1					
tokio-util-0.7.16					
tower-0.5.2					
universal-hash-0.5.1					
url-2.5.7					
utf8_iter-1.0.4					
want-0.3.1					
windows-registry-0.5.3					
winreg-0.55.0					
rustc-demangle-0.1.24					
Rustc-demangle-0.1.26					
Date	08.01.206		10.01.2026	08.01.206	08.01.206

This makes deviations immediately visible without having to manually diff text dumps.

Compile timestamps

Before grouping anything, I also checked when the individual binaries were compiled.

This can be extracted using:

```
for x in *.exe; do printf "$x"; objdump -x "$x" | grep "Time/Date"; don
```

Result:

File Hash	Compile Date
ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914	Tue Sep 23 10:44:52 2025
e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd	Sun Oct 19 21:17:34 2025
e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108	Sun Nov 16 23:41:42 2025
a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79	Sun Nov 16 23:41:42 2025
7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58	Mon Jan 5 06:33:07 2026

TLSH similarity analysis

As a final preprocessing step I generated **TLSH hashes** for all samples.

TLSH is a fuzzy hashing algorithm that does not aim for exact equality, but instead quantifies structural similarity between files. Unlike cryptographic hashes, TLSH allows us to estimate how closely two binaries resemble each other internally even if they differ in resources, padding, timestamps or small code changes.

For our malware analysis purpose, this is especially useful when clustering variants, detecting rebuilds and separating development branches.

To make the distances reproducible, I wrote a small script that computes TLSH values directly from the binaries and performs pairwise comparisons:

```
=== TLSH SIMILARITY RESULTS (lowest diff first) ===
 5 a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79.exe <-> e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe
31 7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58.exe <-> e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe
35 7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58.exe <-> a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79.exe
35 a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79.exe <-> ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914.exe
35 ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914.exe <-> e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe
43 7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58.exe <-> e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd.exe
47 7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58.exe <-> ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914.exe
68 e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd.exe <-> e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108.exe
72 a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79.exe <-> e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd.exe
72 ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914.exe <-> e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd.exe
```

Clustering the samples

With all of this data available, we can now perform a reasonably robust clustering of the samples

The key point here is that no single metric is deciding on its own. Instead, the correlation of multiple independent signals matters:

- TLSH similarity
- Compile timestamps
- Library fingerprints
- Embedded user artefacts

Cluster 1: Core codebase

The first cluster represents the actual core of the codebase and contains the samples:

- e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd
- e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108
- a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79

All three share identical lib dependencies and are compiled close together in time, two of them even share the exact same compile timestamp. One sample also contains the user artefact "**Bruno**", which further hints to a consistent build environment.

Despite moderate TLSH distance to some variants, the internal similarity inside this cluster remains high enough to confidently assume a shared codebase.

Cluster 2: Early feature branch

A second, clearly separable cluster consists solely of:

ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914

This sample extends the otherwise stable library stack with an additional `winreg` dependency while remaining TLSH-close to the core samples. Combined with its earlier compile timestamp, i think this a temporary feature branch or experiment rather than a fully independent development line.

Cluster 3: Later modernization branch

The third cluster is represented by:

7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58

The build differs by multiple additional libraries and introduces a newer version of `rustc-demangle`. Additionally, a different username ("`Jacob`") appears in the artefacts. While the TLSH distance still places this sample within the same family, it consistently sits outside the tighter core cluster range. Taken together, this strongly indicates a later development branch with an updated toolchain and likely expanded functionality

Overall, this results in a three-tier model:

- A stable core cluster forming the technical baseline
- An early feature branch with minimal extension
- A later modernization branch with a changed toolchain and build environment

My clustering is not driven by hash similarity alone, but only becomes reliable through the combined temporal, structural and artefact based correlation. It also provides insight into development practices, build discipline and potential role separation within the development process of the malware

Version mapping

Based on this clustering, I derived the following version mapping:

Version	Sample (SHA256)	Compile Date	Cluster	Besonderheiten	Developer
v0.9	ddc6e6c76ac325d89799a50dff11ec69ed3b5341740619b8e595b8068220914	23.09.2025	Registry-Branch	+ winreg library (Feature-Experiment)	unknown
v1.0	e081bc408f73158c7338823f01455e4f5185a4365c8aad1d60d777e29166abbd	19.10.2025	Core	Baseline Toolchain	Bruno
v1.1	e61b2ed360052a256b3c8761f09d185dad15c67595599da3e587c2c553e83108	16.11.2025	Core	Parallel Build, stabiler Stack	Bruno
v1.1b	a2001892410e9f34ff0d02c8bc9e7c53b0bd10da58461e1e9eab26bdbf410c79	16.11.2025	Core	Parallel Build (gleicher Timestamp)	Bruno
v2.0	7523e53c979692f9eecff6ec760ac3df5b47f172114286e570b6bba3b2133f58	05.01.2026	Jacob-Branch	Neue Libs + neue rustc-demangle	Jacob

At this point we already have a solid structural overview without having touched the actual code yet. This part comes next.

For the code comparison I use Ghidras Version Tracking module. The versions will be analyzed in the following pairs:

v0.9 <-> v1.0
v1.0 <-> v1.1
v1.1 <-> v2.0

So three comparisons in total which honestly is still a very manageable workload

Let's get started

Version Tracking workflow

I won't go into a full tutorial on how Ghidras Version Tracking works. Instead, I'll briefly outline my simple workflow for today.

First, I set the confidence filter to:

-1.0 -> 0.950

Everything above that can usually be assumed identical across samples. I select all those findings and accept them in bulk.

For the remaining findings, I set up a small quality-of-life shortcut:

SHIFT + T = Assign Tag

Then I filter the results by:

- Tag (ascending)
- Type (ascending)
- Status (ascending)

Tag	3	Sessio...	Status	2	Type	1	Score
		6			Function		1.000
		13			Function		1.000
		13			Function		1.000
		6			Function		1.000

and tighten the confidence filter slightly to:

-1.0 → 0.95

This gives me a clean view of all unresolved function-level differences without assigned tags, exactly the changes that matter most when identifying structural evolution.

It looks roughly like this:

The screenshot displays the Version Tracking application interface. The top window, titled 'Version Tracking Matches', shows a table of function-level differences. The table has columns for Tag, Session, Status, Type, and Score. The bottom window, titled 'Function Comparison Provider', shows a code comparison view. The left pane displays the source code, and the right pane displays the destination code. The code is in C++ and shows function definitions and calls. The interface includes a filter bar at the bottom of the matches table, allowing users to filter by score, confidence, and length. The code comparison view includes a 'Decompiler View' and a 'Function Graph View'.

The workflow is simple:

- Select the next finding
- Open code comparison (SHIFT + C)
- If identical -> accept (SHIFT + A)
- If different -> assign a tag for later analysis (SHIFT + T)

Once accepted or tagged, the finding disappears from the filtered view and I repeat the process.

I iterate this until all relevant changes are reviewed and documented.

RustyStealer: Evolution from v0.9 to v2.0

What changed, what failed and what the developer learned

During a longitudinal analysis of multiple RustyStealer builds, four development stages become visible for me: an early baseline version (v0.9), a cleanup iteration (v1.0), a short-lived performance refactor (v1.1) and a more fundamental architectural shift in v2.0.

Rather than showing linear feature growth, the samples show a realistic iterative engineering process with experimentation, rollback and eventual consolidation toward stealth and reliability.

This analysis focuses on behavioral deltas inferred from import-level changes and binary artifacts.

Version 0.9: Early baseline and rough edges

The earliest observed build (v0.9) represents a relatively raw implementation.

Notable characteristics:

- Dependency artifacts included unused crates such as `winreg` / `windows-registry` (i guess early experimentation or leftover scaffolding)
- No meaningful behavioral differences compared to later v1.0 builds
- No advanced memory manipulation or cryptographic primitives observed
- Limited internal abstraction and minimal architectural complexity

From a development POV, v0.9 looks like a staging build that still contained residual dependencies and experimental code paths that were not yet cleaned up.

Version 1.0: Cleanup and stabilization

The transition from v0.9 to v1.0 did not introduce meaningful functional changes. The most visible difference was the removal of unused dependency artifacts (such as the `winreg` crate) without any observable change in imports or runtime behavior.

This looks more like a dependency cleanup rather than a behavioral refactor

Key observations:

- No changes in imported Windows APIs
- No observable change in capabilities or execution flow
- Binary differences consistent with rebuild noise and dependency graph cleanup

This phase reflects basic project hygiene rather than feature development.

Version 1.1: Performance experiment and internal refactor

Version 1.1 marks the first technically ambitious change

Major changes observed:

Introduction of asynchronous and overlapped file I/O:

- `ReadFile`, `NtReadFile`
- `GetOverlappedResult`, `Cancellable`
- Event-based synchronization (`CreateEventW`, `WaitForMultipleObjects`)

Migration to modern Windows path resolution via `SHGetKnownFolderPath`.

Explicit memory cleanup for shell APIs using `CoTaskMemFree`.

Removal of host fingerprinting:

- `GetComputerNameExW`
- `GetUserNameW`
- `NetGetJoinInformation`

This version appears to have attempted:

- Faster and more scalable file harvesting.
- Cleaner filesystem abstraction.

- Reduced behavioral noise by temporarily disabling host profiling.

However, asynchronous I/O pipelines significantly increase complexity and often cause instability in realworld environments
This version looks more like a technical experiment rather than a long term production design for me

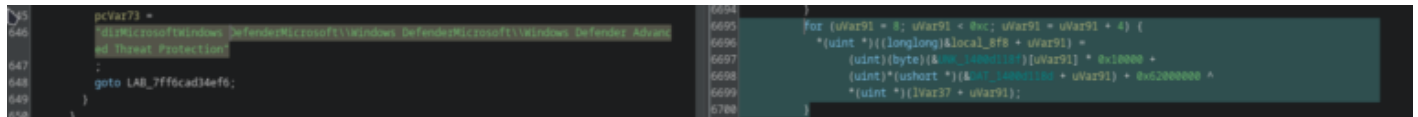
Additional findings from later code diffs

Looking at later builds and comparing them back to this branch reveals several important follow-up changes that appear to originate from the 1.1 refactor.

String handling and Defender artefacts

Older builds still expose cleartext strings such as direct Windows Defender directory paths, making static detection trivial.

In later builds these strings disappear entirely and are reconstructed at runtime using multiple XOR-based decoding routines and different key constants.

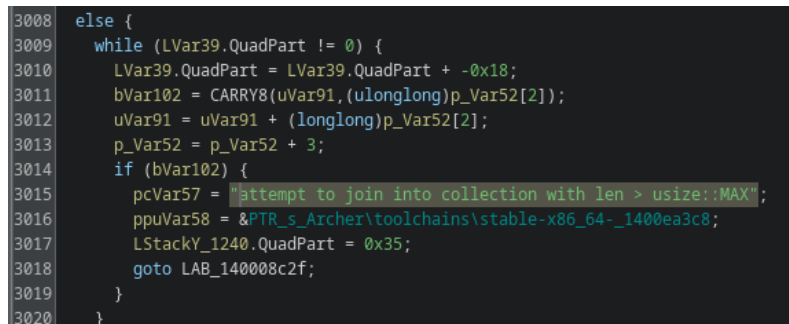


I believe the outcome of the 1.1 refactor was the realization that static strings were becoming a liability, leading to systematic runtime string decryption in subsequent versions.

Collection handling and stability

Later builds include additional safety checks when aggregating dynamic collections (explicit overflow protection when joining result sets)

The same logic exists in earlier builds but is less defensive and less structured.

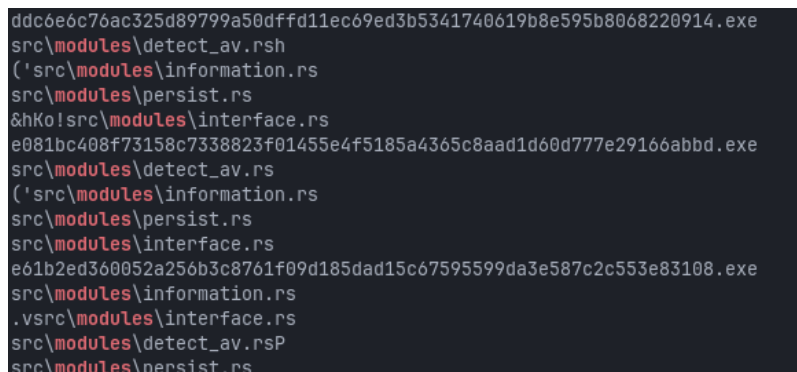


This points to stability issues encountered when processing larger data sets, which maybe were discovered during or shortly after the scalability experiments introduced in version 1.1.

Modular architecture continuity

Both older and newer builds continue to reference internal Rust modules such as:

- `src\modules\persist.rs`
- `src\modules\interface.rs`
- `src\modules\information.rs`
- `src\modules\detect_av.rs`



It confirms that the internal refactor in 1.1 successfully established a modular layout that later versions could iterate on without further structural changes.

Toolchain and build artefacts

Later samples expose more Rust toolchain artefacts (`stable-x86_64` paths and related metadata) and show noticeable hash divergence despite similar functionality.

This looks like a change in compiler version or build environment after this refactor phase, which may also explain subtle behavioral and binary layout differences between closely related samples.

My take

Version 1.1 looks like the point where the developers started stress testing their own architecture.

They might have pushed performance and concurrency, simplified some noisy host profiling and cleaned up API usage.

The follow-up changes visible in later diffs show that this experiment exposed two main pain points:

- Static strings and predictable artefacts were too easy to signature.
- Increased scale and concurrency exposed stability issues in collection handling.

Instead of doubling down on complex async I/O, later versions appear to focus more on obfuscation, robustness and operational reliability, while keeping the modular structure introduced here.

imo 1.1 feels less like a production release and more like a learning phase that shaped the technical direction of everything that followed.

Version 2.0: Architectural consolidation and stealth optimization

Version 2.0 represents a clear update in development priorities.

Instead of further expanding the async pipeline, the developer reverted many of the experimental components and focused on stealth, operational control and internal maturity.

Major changes observed:

Removal of asynchronous file pipeline

All overlapped I/O and event-based sync primitives present in v1.1 have been fully removed in v2.0 and later builds. No replacement async scheduling mechanisms (IOCP, APCs, worker pools or deferred execution patterns) are observable in the call graph.

The runtime execution flow is now strictly sequential, with simplified synchronization primitives replacing the earlier concurrency model.

It looks like the experimental async pipeline was abandoned in favor of a more predictable and operationally stable execution model.

It reduces architectural complexity, lowers race condition risk and improves long-term stability across victim environments.

All overlapped I/O and event-based synchronization APIs were removed, indicating that the v1.1 experiment was abandoned in favor of a simpler and more predictable execution model.

Reintroduction of host fingerprinting

Host identification APIs returned:

- `GetComputerNameExW`
- `GetUserNameW`
- `NetGetJoinInformation`


```

41 else {
42     local_38 = 0;
43     puStack_50 = (undefined *)0x0;
44     puStack_50 = local_50[0];
45     local_30 = 0xffffffff50e0;
46     local_50[0] = (HANDLE)0x0;
47     uVar3 = 0x0100000;
48     if (param_2 != 0) {
49         uVar3 = 0x0100000;
50     }
51     local_30 = local_38;
52     iVar1 = NtCreateNamedPipeFile
53         (local_50, uVar3, local_30, local_60, param_2 + 1, 2, 0, 0, 0, 1, 0x10000, 0x10000,
54          local_30);
55     hObject_80 = local_50[0];
56     if (iVar1 < 0) {
57         lVar2 = RtlNtStatusToDosError(iVar1);
58         param_1[1] = lVar2 << 0x20 | 2;
59         *param_1 = 1;
60     }
61     else {
62         puStack_50 = local_50[0];
63         uStack_80 = uStack_80 | 2;
64         local_50[0] = (HANDLE)0x0;
65         uVar3 = 0x0100000;
66         if (param_2 != 0) {
67             uVar3 = 0x0100000;
68         }
69         iVar1 = NtOpenFile(local_50, uVar3, local_30, local_60, 0, 0x60);
70         if (iVar1 < 0) {
71             lVar2 = RtlNtStatusToDosError(iVar1);
72             param_1[1] = lVar2 << 0x20 | 2;
73             *param_1 = 1;
74             CloseHandle(hObject_80);
75         }
76         else {
77             param_1[1] = hObject_80;
78             param_1[2] = local_50[0];
79             *param_1 = 0;
80         }
81     }
82     CloseHandle(hObject);
83 }
84 return;

```

```

162 local_88 = local_60;
163 local_49 = 1;
164 local_4a = 0;
165 /* WARNING: Subroutine does not return */
166 FUN_1400cb4ba(local_50, 5Var12, &PTR_DAT_140110ea);
167 LAB_1400a7a0:
168 hObject = CreateNamedPipeW(lpName, local_3c, local_34, 1, 0x10000, 0x10000, 0, (LPSECURITY_ATTRIBUTES)0x0
169 );
170 unaff_R8X = local_88;
171 unaff_R13 = local_68;
172 if (hObject != (HANDLE)0xffffffffffff) {
173     if (uVar9 != 0) {
174         FUN_1400ba740(lpName, 2);
175     }
176     local_e0 = (undefined *)(((ulonglong)local_e0 & 0xffffffff00000000);
177     local_c8 = CONCAT71(CONCAT61(((int6)local_c8 >> 0x10), local_4b), local_4b) & 0xffff00000000ffff &
178         1;
179     local_d0 = (DWORD *)0x0;
180     uStack_c8 = 0;
181     local_100 = (DWORD *)CONCAT44(local_100, 4, 4, 0x18);
182     local_f8 = (code *)0x0;
183     local_f8 = (undefined *)CONCAT44(local_f8, 4, 4, 1);
184     local_d8 = local_100;
185     local_88 = unaff_R13;
186     local_88 = unaff_R8X;
187     local_49 = 1;
188     local_4a = 1;
189     lVar5 = local_100;
190     local_118 = hObject;
191     lVar8 = FUN_1400a7a0(unaff_R13, local_100, local_e0);
192     if (lVar8 == 0) {
193         local_90[1] = hObject;
194         local_90[2] = lVar5;
195         *local_90 = 0;
196         if (unaff_R8X == 0) {
197             return;
198         }
199         FUN_1400ba740(unaff_R13, 1);
200         return;
201     }
202     local_90[1] = lVar5;
203     *local_90 = 1;
204     goto joined_r0x000140ba7cd8;
205 }

```

This reduces visibility to user-mode API hooking and increases resistance against common EDR userland instrumentation.

Introduction of cryptographic randomness and entropy hardening

Version 2.0 introduces a dedicated entropy generation pipeline based on [SystemFunction036](#) (RtlGenRandom), replacing the previously implicit and largely deterministic behavior.

Rather than simply requesting random bytes, the implementation builds a full entropy pool that combines cryptographically secure randomness with adaptive **jitter-based sampling** and internal quality validation.

The generated entropy is statistically evaluated at runtime and rejected if it fails basic stability and distribution thresholds.

Only entropy that passes these health checks is propagated further into the execution flow.

```

while ((int)uVar4 != 0x400) {
    uVar21 = uVar4 & 0x1f0;
    uVar3 = uVar4 & 0x10;
    uVar27 = (ulonglong)((int)uVar4 & 0x1f0U & 0x1f0);
    if (uVar4 < 0x200) {
        uVar20 = *(uint *)((longlong)&local_2080 + uVar4 * 4 + 4);
        uVar6 = *(uint *)((longlong)&uStack_2068 + uVar27 * 4);
        uVar10 = (uVar6 << 0x18 | uVar6 >> 8) + *(int *)((longlong)&local_2080 + uVar21 * 4) +
            ((local_2060[uVar27 + 5] << 0x16 | local_2060[uVar27 + 5] >> 10) ^
            (uVar20 << 9 | uVar20 >> 0x17));
        *(uint *)((longlong)&local_2080 + uVar21 * 4) = uVar10;
        uVar6 = *(uint *)((longlong)&local_2070 + uVar27 * 4);
        uVar10 = *(int *)((longlong)&aiStack_1480 + (ulonglong)(uVar6 >> 0xe & 0x3fc)) +
            *(int *)((longlong)&local_1880 + (ulonglong)(uVar6 & 0xff) * 4) ^ uVar10;
        *(uint *)((longlong)&local_2080 + uVar21 * 4) = uVar10;
        uVar6 = *(uint *)((longlong)&uStack_2078 + uVar4 * 4);
        uVar24 = *(uint *)((longlong)&uStack_2078 + uVar4 * 4 + 4);
        uVar26 = *(uint *)((longlong)&uStack_2068 + uVar27 * 4 + 4);
        uVar25 = (uVar26 << 0x18 | uVar26 >> 8) + uVar20 +
            ((local_2060[uVar27 + 6] << 0x16 | local_2060[uVar27 + 6] >> 10) ^

```

The call chain shows that the generated randomness directly feeds into subsequent runtime decoding routines, where multiple obfuscated data blocks are **reconstructed through XOR-based transformations and dynamic key**. In earlier versions, these decode paths relied **almost entirely on static constants and predictable transformations**.

In v2.0, parts of this process become entropy-dependent and therefore non-deterministic across executions

```

for (puVar82 = (undefined8 *)0x0; puVar82 < (undefined8 *)0x10; puVar82 = puVar82 + 1) {
    uVar40 = FUN_140001c6d((byte *)(&lVar92 + (longlong)puVar82));
    *(ulonglong *)(&(longlong)&local_2878 + (longlong)puVar82) =
        uVar40 ^ *(ulonglong *)(&(longlong)&DAT_1400c6e7c + (longlong)puVar82);
}

```

From an operational perspective this has several implications:

- Runtime decoding behavior becomes less reproducible for static analysis and sandbox instrumentation.
- Signature-based detection and clustering become more fragile, as internal state and derived values differ per execution.
- Correlation across samples using hash similarity, instruction-level signatures or deterministic unpacking pipelines becomes significantly harder.
- The malware gains a higher degree of resilience against emulation environments with weak or manipulated entropy sources.

While the mechanism could theoretically support use cases such as object naming, session identifiers or cryptographic material, the integration into the decode pipeline indicates more that the primary motivation is obfuscation hardening and stealth rather than feature expansion

It marks a shift in development priorities: the developers invest in making existing functionality harder to observe, reproduce and classify

Memory protection manipulation (VirtualProtect)

`VirtualProtect` appears in v2.0, but not as a generic “unpacking” indicator.

It is used to flip a read-only dispatch table to writable just long enough to store dynamically resolved API pointers, then lock it back down. That's consistent with the shift towards runtime API resolving and NT-level calls: fewer static imports, less signature surface and a resolver layer that is harder to tamper with or trivial hook

Internal synchronization refactor

Thread state handling was refactored away from TLS primitives and migrated towards modern synchronization APIs (InitializeCriticalSectionEx)

I think it's for architectural cleanup and improved concurrency control.

```
do {
    InitializeCriticalSectionEx((LPCriticalSection)(puVar6 + 0x30), 4000, 0);
    *(undefined **)(&local_2048 + (longlong)DAT_14012a788) = puVar6;
    if (*(longlong *)(&DAT_14012a280)[lVar2 >> 6] + 0x28 + (ulonglong)((uint)lVar2 & 0x3f) * 0x48
        + 2U < 3) {
        *puVar3 = 0xffffffff;
    }
    lVar2 = lVar2 + 1;
    puVar6 = puVar6 + 0x58;
    lVar1 = lVar1 + 8;
    puVar3 = puVar3 + 0x16;
    lVar5 = lVar5 + -1;
} while (lVar5 != 0);
return 0;
```

Obfuscation hardening and decoder abstraction

Compared to version 1.1, version 2.0 further abstracts string decryption and reconstruction logic.

Previously visible XOR decoders with fixed constants and static data offsets were refactored into more generic reconstruction helpers and indirect decode routines.

```
2519 for (pvVar86 = (LARGE_INTEGER *)0x0;
2520      (lVar97 = pvVar49 != (LPVOID)0x0, pvVar49 = (LPVOID)((longlong)pvVar49 + -1), bVar97 &&
2521      (local_2048 != pvVar86)); pvVar86 = pvVar86 + 3) {
2522     FUN_14000e568((SIZE_T *)(&local_2158.QuadPart, *(undefined **)(&local_2158.QuadPart + 8),
2523                 *(SIZE_T *)(&local_2158.QuadPart + 8x18)));
2524     *(ulonglong *)(&pvVar71 + 8x18 + (longlong)pvVar86) =
2525         CONCAT26(uStack_2142,
2526                 CONCAT24(uStack_2144,
2527                         CONCAT13(abStack_2148[0],
2528                                 CONCAT12(abStack_2148[1], abStack_2148[0]))));
2529     *(ulonglong *)(&pvVar71 + (longlong)pvVar86) =
2530         CONCAT26(local_2158.s.HighPart_2_2,
2531                 CONCAT15(local_2158.s.HighPart_1_1,
2532                         CONCAT14(local_2158.s.HighPart_0_1, local_2158.s.LowPart)));
2533     *(ulonglong *)(&local_2158.s.HighPart_0_1 + (longlong)pvVar86) = 0;
2534     CONCAT17(uStack_2149,
2535             CONCAT16(uStack_214a, CONCAT15(uStack_214b, CONCAT14(bStack_214c, uStack_2150))));
2537     lVar46.QuadPart = lVar46.QuadPart + 8x18;
2538 }
```

```
6686 bVar102 = true;
6687 lVar98 = 0;
6688 while (bVar102) {
6689     *(ulonglong *)(&local_8f8 + lVar98) =
6690         ((ulonglong)(byte)(&DAT_140001100)[lVar98] | 0x4573fdb0efebf280) ^
6691         *(ulonglong *)(&lVar37 + lVar98);
6692     bVar102 = false;
6693     lVar98 = 8;
6694 }
6695 for (uVar91 = 8; uVar91 < 8xc; uVar91 = uVar91 + 4) {
6696     *(uint *)(&local_8f8 + uVar91) =
6697         (uint)(byte)(&local_8f8)[uVar91] ^ 0x10000 +
6698         (uint)(ushort *)(&local_8f8 + uVar91) ^ 0x10000 +
6699         *(uint *)(&lVar37 + uVar91);
6700 }
6701 bStack_Rec = (byte *)(&lVar37 + 8xc) ^ 0xef;
6702 local_308.s.LowPart_0_1 = (undefined)local_8f8.s.LowPart;
6703 local_308.s.LowPart_1_1 = (undefined)((uint)local_8f8_0_4 >> 8);
6704 local_308.s.LowPart_2_1 = (undefined)((uint)local_8f8_0_4 >> 8x18);
6705 local_308.s.LowPart_3_1 = (undefined)((uint)local_8f8_0_4 >> 8x18);
```

This significantly reduces static signature stability and increases the effort required for automated clustering and YARA-style detection.

Improved memory lifecycle management and cleanup paths

Version 2.0 introduces more explicit cleanup sequences, callback-based destructors and tighter buffer lifecycle handling.

Multiple teardown paths now explicitly free dynamically allocated buffers, invoke destructor callbacks and normalize internal state after large collection operations.

It might be an effort to reduce memory leaks, crash probability and long-running instability during high-volume data processing.

```

FUN_140029590(local_21d8,pvVar49);
p1Var43 = local_21a8;
if (local_21a8 != (longlong *)0x0) {
    FUN_140029852(local_21a8);
    FUN_14001706b(p1Var43,8);
}
pvVar82 = local_2190;
pvVar49 = local_2198;
if ((code *)local_2190 != (code *)0x0) {
    (*(code *)local_2190)(local_2198);
}
if (pvVar82[1] != 0) {
    FUN_14001706b(pvVar49,pvVar82[2]);
}
}

```

Build pipeline evolution

Build artifacts show a different Rust toolchain and build environment compared to version 1.1.

Toolchain paths shifted from shared “Archer” layouts towards user-local rustup environments, explaining observed TLSH divergence despite largely stable runtime behavior.

14010d418	C:\Users\Jacob\.cargo\registry\src	"C:\Users\Jacob\.cargo\registr...	ds
140102ee0	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400c8403	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400c81f9	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400c8aa0	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400e1c60	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400c8b38	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400e1d20	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400e1bad	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1401131d8	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400dd7a0	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
14010bc98	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds
1400ddb68	C:\Users\Jacob\.cargo\registry\src\index.cr...	"C:\Users\Jacob\.cargo\registr...	ds

This reflects internal “”””build pipeline”””” changes rather than functional malware changes.

Summary

Version 2.0 prioritizes:

- Simplification and rollback of risky experimental I/O designs
- Stronger stealth via native API usage and obfuscation hardening
- Improved runtime stability and memory hygiene
- Reintroduction of host profiling for operational correlation
- Internal architectural consolidation rather than feature expansion

Overall, this release reflects a stabilization phase where operational reliability and detection resistance outweigh rapid feature development

Conclusion

If there’s one takeaway from this exercise, it’s that even relatively “boring” commodity implants become surprisingly interesting once you stop treating them as static blobs and start treating them like evolving software projects.

By correlating build artefacts, dependency drift, fuzzy hashes and targeted code diffs, we can reconstruct a fairly coherent development timeline for RustyStealer without having to reverse every single function into oblivion. We see experimentation phases (hello async pipeline), rollbacks when ideas don’t survive real-world friction, hardening towards stealth (dynamic API resolution, entropy injection) and the usual toolchain churn that comes with real developers shipping real code, even if that code happens to be malware.

None of this magically turns RustyStealer into some nextgen cyber superweapon. **It’s still fundamentally a fairly standard implant doing fairly standard implant things.**

Also it’s a nice reminder that “grep strings > extract a few IOCs > call it a day” only gets you so far. If you actually want to understand how an actor operates, what they experiment with and where their engineering effort goes, you have to look at trends, not snapshots.

Which admittedly takes more time, more coffee and slightly more patience than copy-pasting hashes into VirusTotal buuuut the payoff is a much clearer picture of what’s really going on under the hood.

There’s also a more general lesson in all of this imo

If the goal is to actually understand a threat actor, not just label a sample and move on, then obsessing over individual functions, opcodes and micro-optimizations only gets you part of the way.

That level of detail is useful, but it doesn’t tell you how the tooling evolves, where engineering effort is invested, which ideas get abandoned or how operational priorities shift over time.

These signals live in the truly boring parts: build artefacts, dependency drift, architectural refactors, version churn and the occasional accidental breadcrumb left behind in a binary or wherever.

When you line those up over multiple samples, you start seeing intent and process instead of isolated behaviour and this exactly what we want ^-^