KrustyLoader - About stripped Rust symbol recovery


nofix.re/posts/2024-11-02-rust-symbs/

Feb 10, 2024

> I have been writing a tool to automate most of the work presented in this blog post. It is available here and provided with absolutly no guarantees. This is just me writing some stuff to help with my daily reverse engineering.

Recently, Ivanti disclosed two critical zero-day vulnerabilities affecting their Connect Secure VPN product. It seems that threat actors have been actively exploiting those in the wild.

Synacktiv did an excellent job with a quick dynamic reverse engineering, and gave samples hashes. I wanted to give static analysis a shot to delve deeper into Rust internals, and learn more about Rust reverse engineering.

`Krustyloader` uses Tokio, a popular crate for asynchronous development. Statically reverse engineer such code without any symbols is a challenging task : Tokio has complex code that provides a custom runtime with its own internals tricks to learn about.

I have been interested in recovering symbols from stripped Rust executables for a little while now, and it is not common to find malware written in Rust. This seemed like a good opportunity to work on Rust symbols recovery.

A common theme in reverse engineering is generating signatures for libraries used in your target executable to distinguish library code from non-library code. This is typically done by recompiling those libraries with the same compile options used in the target and signing the result with a suitable tool, such as IDA's sigmake or Binary Ninja's sigkit.

This is how I started my reverse engineering journey of this Rust payload.

Rust artifacts

First step of symbol recovery is knowing which compiler has been used, and what are the dependencies.

A default `cargo build --release` leaves a lot of information into its target executable for us to exploit:

- rustc's GitHub commit hash ;
- each dependency name and version.

These are easily obtainable with a simple `grep`:

```
$ grep -a rustc /tmp/sample
/rustc/90c541806f23a127002de5b4038be731ba1458ca/library/core/src/slice/iter.rsMap
must not be polled after it returned
```

And just like that, we know that rustc's hash used at compile time is : `90c541806f23a127002de5b4038be731ba1458ca`.

Getting toolchain's version

A query to GitHub shows us tags associated with rustc's commit hash. Latest one should be retained, in this case, 1.70.0.

Finding the correct toolchain

We also need to find the toolchain that has been used to compile the target (GNU, MUSL, MinGW…).

I found no magic to easily find this information other than compiling myself executables with different toolchains and compare the results to find heuristics.

In this case, it was straightforward to find the correct toolchain:

```
$ file /tmp/sample
/tmp/sample: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), static-pie
linked, stripped

$ ldd /tmp/sample
        statically linked
```

The target is a statically compiled executable, likely using musl.

I assumed malware authors were lazy, thus opted for a solution that required minimal setup. Checking rust's platform support tiers reveals that `x86_64-unknown-linux-musl` is a "Tier 2 with Host tool" target, which means that it is a "guaranteed to build" toolchain, and is an easy target to build to. Musl looks like the perfect candidate for our toolchain.

Putting the pieces together, we can guess it's likely that the toolchain is `1.70.0-x86_64-unknown-linux-musl`.

Dependencies

A quick CTF-ish `strings | grep` is enough to find dependencies used in our target:

```
$ s=$(strings /tmp/sample); echo "${s//registry/\n}" | grep -a '\.rs' | sort
/rustc/90c541806f23a127002de5b4038be731ba1458ca/library/alloc/src/collections/btree/m
ap/entry.rs [...more garbage...]
/rustc/90c541806f23a127002de5b4038be731ba1458ca/library/core/src/slice/iter.rs
[...more garbage...]
/rustc/90c541806f23a127002de5b4038be731ba1458ca/library/core/src/sync/atomic.rs
[...more garbage...]
/rustc/90c541806f23a127002de5b4038be731ba1458ca/library/std/src/io/mod.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/addr2line-0.17.0/src/function.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/aes-0.7.5/src/soft/fixslice64.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/bytes-1.4.0/src/buf/buf_impl.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/cfb-mode-0.7.1/src/lib.rs [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/futures-channel-0.3.28/src/mpsc/mod.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/futures-util-
0.3.28/src/future/try_future/try_flatten.rs [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/generic-array-0.14.7/src/lib.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/getrandom-0.2.10/src/util_libc.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/gimli-0.26.2/src/read/line.r [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/hashbrown-0.12.3/src/raw/mod.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/hex-0.4.3/src/lib.rs [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/http-0.2.9/src/version.rs [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/httparse-1.8.0/src/iter.rs [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/httparse-1.8.0/src/lib.r [...more garbage...]
/src/index.crates.io-6f17d22bba15001f/hyper-0.14.27/src/body/body.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/miniz_oxide-0.5.3/src/inflate/output_buffer.rs
[...more garbage...]
/src/index.crates.io-6f17d22bba15001f/num_cpus-1.15.0/src/linux.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/rand-0.8.5/src/rngs/thread.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/rand_chacha-0.3.1/src/guts.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/rustc-demangle-0.1.21/src/v0.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/socket2-0.4.9/src/socket.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/tokio-1.29.0/src/util/wake_list.rs [...more
garbage...]
/src/index.crates.io-6f17d22bba15001f/want-0.3.1/src/lib.rs [...more garbage...]
```

> These results have been edited for clarity sake.

And now with a clean formating and a bit of automation:

```
$ rbi info target_sample.bin
TargetRustInfo(
    rustc_version='1.70.0',
    rustc_commit_hash='90c541806f23a127002de5b4038be731ba1458ca',
    dependencies=[
        Crate(name='addr2line', version='0.17.0', features=[], repository=None),
        Crate(name='aes', version='0.7.5', features=[], repository=None),
        Crate(name='bytes', version='1.4.0', features=[], repository=None),
        Crate(name='cfb-mode', version='0.7.1', features=[], repository=None),
        Crate(name='futures-channel', version='0.3.28', features=[],
repository=None),
        Crate(name='futures-core', version='0.3.28', features=[], repository=None),
        Crate(name='futures-util', version='0.3.28', features=[], repository=None),
        Crate(name='generic-array', version='0.14.7', features=[], repository=None),
        Crate(name='getrandom', version='0.2.10', features=[], repository=None),
        Crate(name='gimli', version='0.26.2', features=[], repository=None),
        Crate(name='hashbrown', version='0.12.3', features=[], repository=None),
        Crate(name='hex', version='0.4.3', features=[], repository=None),
        Crate(name='http', version='0.2.9', features=[], repository=None),
        Crate(name='httparse', version='1.8.0', features=[], repository=None),
        Crate(name='hyper', version='0.14.27', features=[], repository=None),
        Crate(name='miniz_oxide', version='0.5.3', features=[], repository=None),
        Crate(name='num_cpus', version='1.15.0', features=[], repository=None),
        Crate(name='rand', version='0.8.5', features=[], repository=None),
        Crate(name='rand_chacha', version='0.3.1', features=[], repository=None),
        Crate(name='socket2', version='0.4.9', features=[], repository=None),
        Crate(name='tokio', version='1.29.0', features=[], repository=None),
        Crate(name='want', version='0.3.1', features=[], repository=None)
    ],
    guessed_toolchain='linux-musl',
    guess_is_debug_build=False
)
```

Building the dependencies

First, we have to install the toolchain:

```
$ apt-get install musl musl-dev musl-tools
$ rustup target add x86_64-unknown-linux-musl
$ rustup install 1.70.0-x86_64-unknown-linux-musl
```

Second, we have to download each dependency and compile it. An issue I faced when compiling those is that I wanted to be able to automate the process on both Linux and Windows, using IDA as a signature provider.

I quickly faced an issue since IDA/Hey-Rays' `pcf.exe` only reads static `.lib` libraries (I guess `pelf` works the same).

Compiling `staticlib` Windows files from Rust didn't seem to be an easy task to automate, according to various linkage GitHub issues and StackOverflow posts I found across the internet.

There might be ways of automatizing the build of Rust static libraries on Windows, but it seemed like a tedious and error prone thing to do.

Instead, I choosed to use the great tool idb2pat from Mandiant, which, *well*, turns an IDB to a `.pat` file, which is pattern file format used by IDA tools. This replaces the need of dealing with `pcf.exe`/`pelf` complaining about file format of my compiled libraries.

I will be compiling every dependency as dynamic libraries.

To do so, we need to ensure the following lines exists in the `Cargo.toml` files of each dependency we're going to build:

```
[lib]
crate-type = ["dylib"] # Dynamic library
```

Since we want to compile as much of the available code in the crate as possible, I will be enabling all features when possible:

```
rustup run 1.70.0-x86_64-unknown-linux-musl cargo build --release --lib --all-features
```

> Given that compiling with `--all-features` can lead to failures due to nightly or unstable features on stable toolchains, I've incorporated some intelligence into the automation to attempt compiling with as many features as possible.

Building with musl

Building with musl has constraints. First, it requieres getting MUSL compiled libraries, and linking against them (which can be achieved by adding their location to `LD_LIBRARY_PATH` if you are lazy enough not to deal with `ldconfig`).

Also, musl doesn't support building `dylib` crate type unless you add the `target-feature=-crt-static` compile flag, which disables the need of having a static runtime in the target lib. You can do such a thing by exporting the following environnement variable: `RUSTFLAGS='-C target-feature=-crt-static'`.

Since this defeats the purpose of building with a MUSL toolchain, I found it more convenient to build with the GNU toolchain (`1.70.0-x86_64-unknown-linux-gnu`). It turned out that it didn't make much a difference anyway, as far as I could see in the generated assembly code.

Building the signature

Once every library has been built, we should be able to generate `.pat` files, using the previously mentionned Mandiant's `idb2pat`:

```
idat64 -S./idb2pat.py "/tmp/generated/tokio-1.29.0/target/release/libtokio.so" -a -A
tokio.pat
```

> Note that idb2pat has been slightly modified here for automation purposes.

Repeat that for every dependency, add salt, pepper and a little bit of patience, and we're now set up to use `sigmake`, which is IDA's tool to turn `.pat` files to a signature file that IDA can load:

```
sigmake -n"Test_signature" libtokio.pat [other .pat files] test.sig
```

This should have been the nice, peacefull and calm journey into recovering symbols from stripped Rust executables.

But things did not work out as expected.

Troubles, as all projects have

One thing I *almost* omitted to mention untill now is that we need to compile with the same compilation options that the malware author used to compile its executable.

A fair guess would be that the attacker wasn't against a bit of optimization and speed, thus built in release profile mode.

It's our job to guess whether the default profile mode has been tweaked by the author or not.

As a reminder, the default release profile options are:

```
[profile.release]
opt-level = 3
debug = false
split-debuginfo = '...'  # Platform-specific.
strip = "none"
debug-assertions = false
overflow-checks = false
lto = false
panic = 'unwind'
incremental = false
codegen-units = 16
rpath = false
```

Back to the malware now. One thing suprized me when I first downloaded this rust-based payload was its size:

```
$ du -h /tmp/sample
860K    /tmp/sample
```

Rust's elf bins are knowingly quite big by default (mainly due to symbols, which are absent in release build, but still). Knowing that our target is a static executable, having such a small executable could indidacte that it's likely that the default release profile has been modified by the author.

And it was.

Finding the right release profile

Fiding the right release profile options wasn't a hard task.

There isn't many options in the profile that could modify target size. Candidates are:

- `opt-level` (probably `3`, `s` or `z`)
- `lto` (`thin` or `fat`)
- `codegen-units`

Trying a combinaison of all of them could be an option. But I assumed that the authors were lazy, and that they probably copy/pasted some code over the internet that claims to reduce the code size.

Sure enough, after a bit of googling around I found this:

```
[profile.release]
strip = true # strip symbols from the binary
opt-level = 'z' # Optimize for size
lto = true # Link Time Optimization (LTO)
codegen-units = 1 # Parallel Code Generation Units
```

If you are wondering how `codegen-units` can affect code optimization, so do I. Turns out the author of the GitHub repo has an explaination:

> "Code generation units greater than 1 specify that the compiler toolchain is allowed to process our code in parallel. This most of the time leaves us with unused optimization potential, in this case for size, since it will look at our code in parallel. By default when not specified it is set to 16, which increases compilation speed."

A good way to know if you found the correct compilation options is to compile an hello-world crate yourself with the same toolchain and comparing your target's code in IDA with hello-world's one.

And sure enough, my hello-world code and target one matched.

Introducing: Link Time Optimization

A ***very*** annoying option used in the dropper's compilation profile is `lto = true`. `lto` stands for `Link Time Optimization`. Setting it to `true` means that some optimization can occur *at link time* (after compilation of our dependencies). This is an issue because we signed dependencies independently, without them being actually used in any code. However, the malware author did not do this; all the code for the executable has been compiled at once, meaning that every dependency was built, linked to the target executable, and optimized afterwards.

If any function gets optimized, even a little bit, our library signature will no longer be effective.

A good illustration of this occurs at the beginning of Tokio's runtime. When compiled with options described above, Tokio's runtime starts by calling `RngSeed::new()`, resulting in the following assembly being generated in `libtokio.so`:

```
push    rbp
push    rbx
sub     rsp, 58h
lea     rdi, off_2E60F8
call    std::thread::local::LocalKey$LT$T$GT$::with::hef935abace049e26
mov     rcx, 736F6D6570736575h
xor     rcx, rax
mov     rsi, 646F72616E646F6Dh
xor     rsi, rdx
mov     rdi, 6C7967656E657261h
xor     rdi, rax
mov     r8, 7465646279746573h
xor     r8, rdx
```

While `KrustyLoader` has the following assembly:

```
push    rbp
push    rbx
sub     rsp, 58h
; <some bad boy disappeared here !>
call    sub_4B224
mov     rcx, 736F6D6570736575h
xor     rcx, rax
mov     rsi, 646F72616E646F6Dh
xor     rsi, rdx
mov     rdi, 6C7967656E657261h
xor     rdi, rax
mov     r8, 7465646279746573h
xor     r8, rdx
```

This messes with our signature, and the function does not get recognized.

Monomorphization

Monomorphization is a terrible word to spell.

It also is a word that can be used to describe Rust's generics type system.

Explaining how Rust generics work isn't the purpose of this blog post. If you are not familiar with it, I encourage you reading this entry and this one from fasterthanlime's blog. His blog is just a great place to learn about Rust and other tech stuff in general. If you are of the curious kind, you should check it out.

The main issue with signing generics is that if no one implements your generic, let's say struct `Container<T>`, the compiler just wont have any code to produce for it. This is an issue for libraries generics: if no one uses the `Container<T>` generic type you have in your lib, then it isn't something you will be able to sign. You need someone implementing it, preferably in various ways, to maximise the chances that the generated assembly code you will sign matches the code of your target.

Dirty solution

While trying to solve LTO and Monomorphization issues, a friend and I came up with a somewhat messy solution. Surprisingly, it's been quite effective with popular well-maintained crates like `tokio` and `hyper`.

The idea is to compile tons of code that implements generics of a given dependency in various ways. This should improve the likelihood of encountering code that is:

- optimized similarly to our target (solving LTO's issue) ;
- implementing generics the library exposes (solving monomorphization issue).

I clearly don't want to download and build any random crate from the internet using `Tokio`, since it is quite easy for an attacker to craft a crate that would run arbitrary code at compile time.

A better approach would be compiling `tests`, `examples` and `benches` targets from the dependency. Those should create executables that use most of functions exposed by the crate, while still being optimized by `lto`.

Three issues with that.

First: depending on the crate, `tests`, `examples` or `benches` might not be present, but hey, this is better than nothing.

Second: this approach generates a tons of executables (e.g >200 for tokio), which can be really long to sign, depending on your hardware and the signing tool you are using.

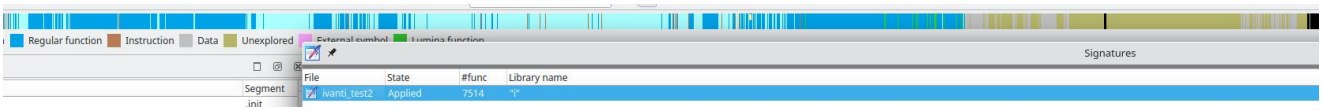
Last but not least: crates from `crates.io` don't always embeed tests, examples and benches, but the project hosted on GitHub often does.

Thus we need to pull the dependency from GitHub, guess the correct branch and tag corresponding to the version used in our target (given that such a branch still exists) and compile everything.

The good part is that giving the GitHub repository URL of a crate is mandatory for publishing something to `crates.io`. Unfortunately, no one is forced to give a valid URL. One could also delete or move the GitHub repository after publication.

Results

The results were quite good. After 15mn of compilation, I was able to generate 1170 executables for IDA to sign, which took another 5-10mn, and matched 7514 functions:



Signing Rust's stdlib

Rust provides everything we need to sign the standard library used by our toolchain.

Toolchain debug information is available to download through `rustup`:

```
rustup component add rustc-dev --toolchain 1.70.0-x86_64-unknown-linux-musl
```

This command downloads symbols, object files and shared libraries (with symbols) used in your toolchain. They are available under `${HOME}/.rustup/toolchains/{toolchain}/`.

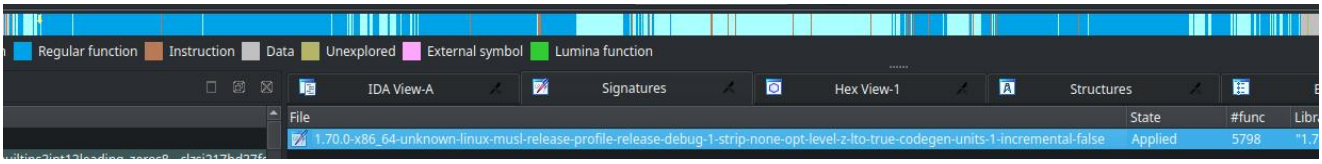
```
$ find ~/.rustup/toolchains/1.70.0-x86_64-unknown-linux-musl/ -type f \( -name "*.so"
-o -name "*.o" \)
./lib/libstd-a9e9fdf6bd876a9c.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crtn.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crtend.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/rcrt1.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/Scrt1.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crtbeginS.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crtbegin.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crt1.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crtendS.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/self-contained/crti.o
./lib/rustlib/x86_64-unknown-linux-musl/lib/libzerofrom_derive-43985d6c6f2d6449.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libchalk_derive-eaa4f39338ccf84d.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libserde_derive-310f743814410e38.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libcstr-6c2f66117dc9abc1.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/librustc_macros-164f925dc05bb4c7.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libtracing_attributes-15ce531b9da618ae.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libicu_provider_macros-
31041477eb05c7f4.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libstd-a9e9fdf6bd876a9c.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/librustc_driver-65a88990b6151b4f.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libthiserror_impl-346d3314154f3ab8.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libzerovec_derive-e1fe9a99c61ac1d4.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libtest-52924d8e151e3987.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libdisplaydoc-a6b6f4c161687932.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libproc_macro_hack-10b1ed1a0a1bb93c.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libyoke_derive-5dbc7091c4175f58.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libunic_langid_macros_impl-
e3626691e4b3859c.so
./lib/rustlib/x86_64-unknown-linux-musl/lib/libderive_more-5e542ae59d78db58.so
./lib/librustc_driver-65a88990b6151b4f.so
./lib/libtest-52924d8e151e3987.so
```

Those are pre-built and can be used to generate a signature, the same way we generated a signature for dependencies.

Results

The best thing about signing stdlib is that it is constant and should not vary depending on the target's compile option.

Here is the result I had signing `1.70.0-x86_64-unknown-linux-musl`:



And here is a tiny example of what both signatures applied looks like in IDA's decompialtion.

Before:

```
sub_497FC(v318, *((_QWORD *)&v318 + 1));
a1[47] = *(_QWORD *)&v342[16];
*(_OWORD *)(a1 + 45) = *(_OWORD *)v342;
sub_36170(v368, &unk_89256, 200LL);
v63 = sub_1511B(v368);
v294 = sub_4AF41(v63, v64);
v297 = v65;
v67 = (_BYTE *)sub_158C9(1LL, 1LL, v65, v66);
*v67 = -13;
v70 = (_BYTE *)sub_158C9(1LL, 1LL, v68, v69);
*v70 = -113;
v71 = 1;
v72 = 0LL;
while ( (v71 & 1) != 0 )
{
  v73 = *(_BYTE *)sub_24AA3(v67, 1LL, v72);
  v74 = (_BYTE *)sub_24D8A(v70, 1LL, v72, &off_2CF178);
  *v74 ^= v73;
  v71 = 0;
  v72 = 1LL;
}
```

After:

```
core::ptr::drop_in_place<alloc::vec::Vec<(gimli::common::DebugInfoOffset,gimli::common::DebugArangesOffset)>>(
  v309,
  *((_QWORD *)&v309 + 1));
a1[47] = *(_QWORD *)&v333[16];
*(_OWORD *)(a1 + 45) = *(_OWORD *)v333;
core::str::converts::from_utf8(v359, &word_89256, 200LL);
v56 = sub_1511B(v359);
v285 = core::str::<impl str>::trim_matches(v56, v57);
v288 = v58;
v60 = (_BYTE *)alloc::alloc::exchange_malloc(1LL, 1LL, v58, v59);
*v60 = -13;
v63 = (_BYTE *)alloc::alloc::exchange_malloc(1LL, 1LL, v61, v62);
*v63 = -113;
v64 = 1;
v65 = 0LL;
while ( (v64 & 1) != 0 )
{
  v66 = *(_BYTE *)<alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index(v60, 1LL, v65);
  v67 = (_BYTE *)<gimli::read::endian_reader::EndianReader<Endian,T> as core::ops::index::Index<usize>>::index(
                   v63,
                   1LL,
                   v65,
                   &off_2CF178);
  *v67 ^= v66;
  v64 = 0;
  v65 = 1LL;
}
```

Articles cited

Thanks to Mandiant's idb2pat which has proven to be a very usefull tool.