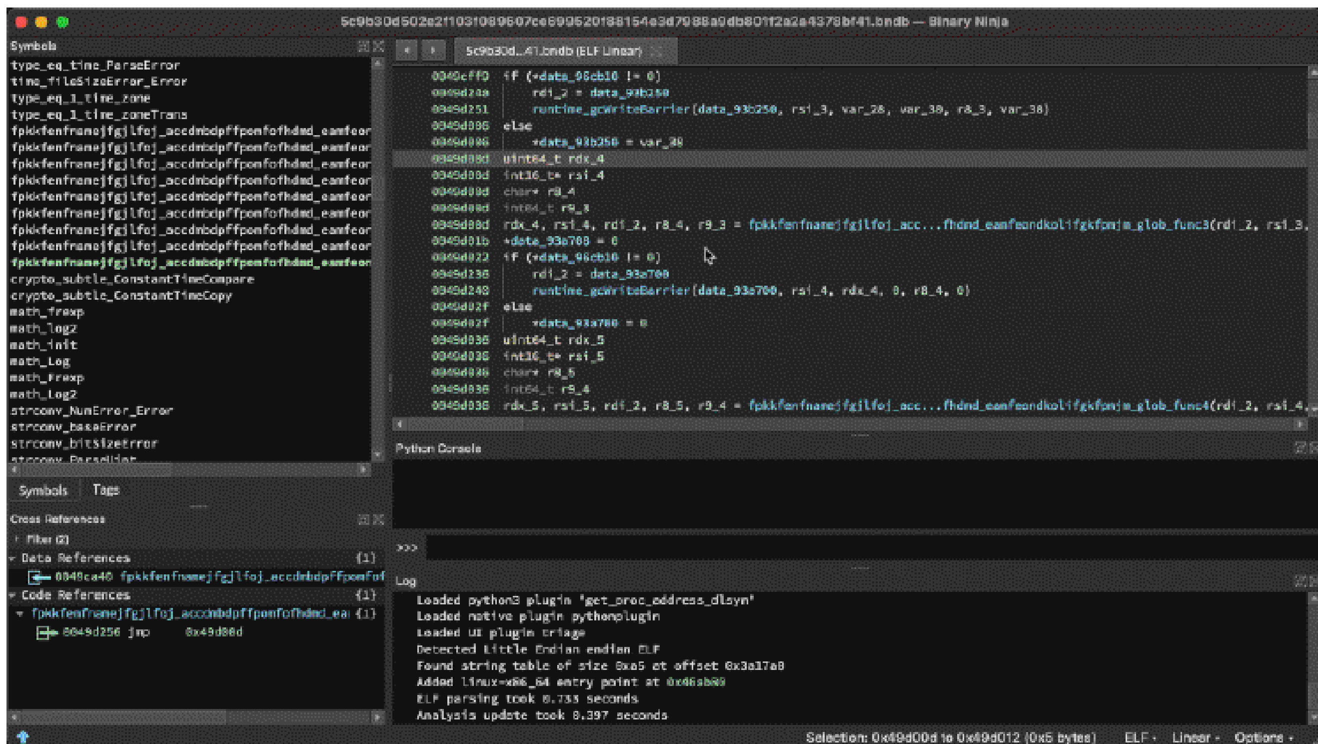
Automated string de-gobfuscation

kryptoslogic.com/blog/2020/12/automated-string-de-gobfuscation/



Authored by: Jamie Hankins on Wednesday, December 2, 2020

Last week the Network Security Research Lab at 360 released a blog post on an obfuscated backdoor written in Go named Blackrota. They claim that the Blackrota backdoor is available for both x86/x86-64 architectures which is no surprise given how capable Golang's cross compilation is.

For the last 4 years we have been using Golang for our internal services, and I can definitely see the allure that Golang has for malware authors:

- Statically compiled binaries by default
- Cross compilation is often as simple as setting two environment variables
- Strong package ecosystem allowing you to pull in code that you need from other sources
- No runtime dependencies
- Esoteric runtime with a non-standard calling convention breaks most decompilation tools forcing reverse engineers to read assembly

Blackrota uses gobfuscate to obfuscate their source code before it gets compiled by the Go toolchain. Gobfuscate presents a number of challenges to reverse engineers but the one I'll be focusing on today is string obfuscation.

String Obfuscation

Malware has been using XOR encoded strings for years now, but Blackrota takes this a step further[1]. It generates a random XOR key per string and wraps the string in a function that XORs the string at runtime to return the correct one.

gobfuscate runs before the compilation process to produce an obfuscated version of your code which is then compiled by the Go compiler:

Before:

```
package main

import "fmt"

func main() {
	fmt.Println("Hello world!")
}
```

After:

```
package main

import "fmt"

func main() {
	fmt.Println((func() string {
		mask := []byte("\x21\x0f\xc7\xbb\x81\x86\x39\xac\x48\xa4\xc6\xaf")
		maskedStr :=
[]byte("\x69\x6a\xab\xd7\xee\xa6\x4e\xc3\x3a\xc8\xa2\x8e")
		res := make([]byte, 12)
		for i, m := range mask {
			res[i] = m ^ maskedStr[i]
		}
		return string(res)
	}()))
}
```

You can use a tool like GCHQ's CyberChef to verify that the result of those 2 byte arrays XORed together is: `Hello world!`.

While you *could* go through the effort of manually XORing each string in a binary you're reversing, it'll get tedious very quickly.

How do I solve it *at scale*?

So I know that Golang has a very capable cross compiler and that I want to deal with these XORed strings across different architectures, but how?

Binary Ninja (aka Binja) has a very powerful intermediate language (IL), which allows us to operate on a representation of a function regardless of the system architecture (assuming there is an architecture loader for it)[2].

When deciding on an approach here I specifically wanted to use Binary Ninja's IL as a learning exercise. The solution could certainly be implemented using any number of alternative tools (e.g. Unicorn), however Binja has the benefit of being very easy to use on any of its supported platforms. Another benefit, of course, is that the IL eliminates the need to understand the target architecture at all, which (depending on implementation) may not be the case using something like Unicorn.

So the solution will need a way of identifying what functions we want to extract the strings from (Candidate Identification) and a way to extract the correct string from Binary Ninja's low level IL.

Candidate Identification

Luckily for us, the Go compiler doesn't inline this function call so it's seen as an entirely new function:

Assuming the approach I take can avoid the Go runtime functions: `_runtime.morestack_noctxt` and `_runtime.slicebytetostring` it should be pretty easy to emulate.

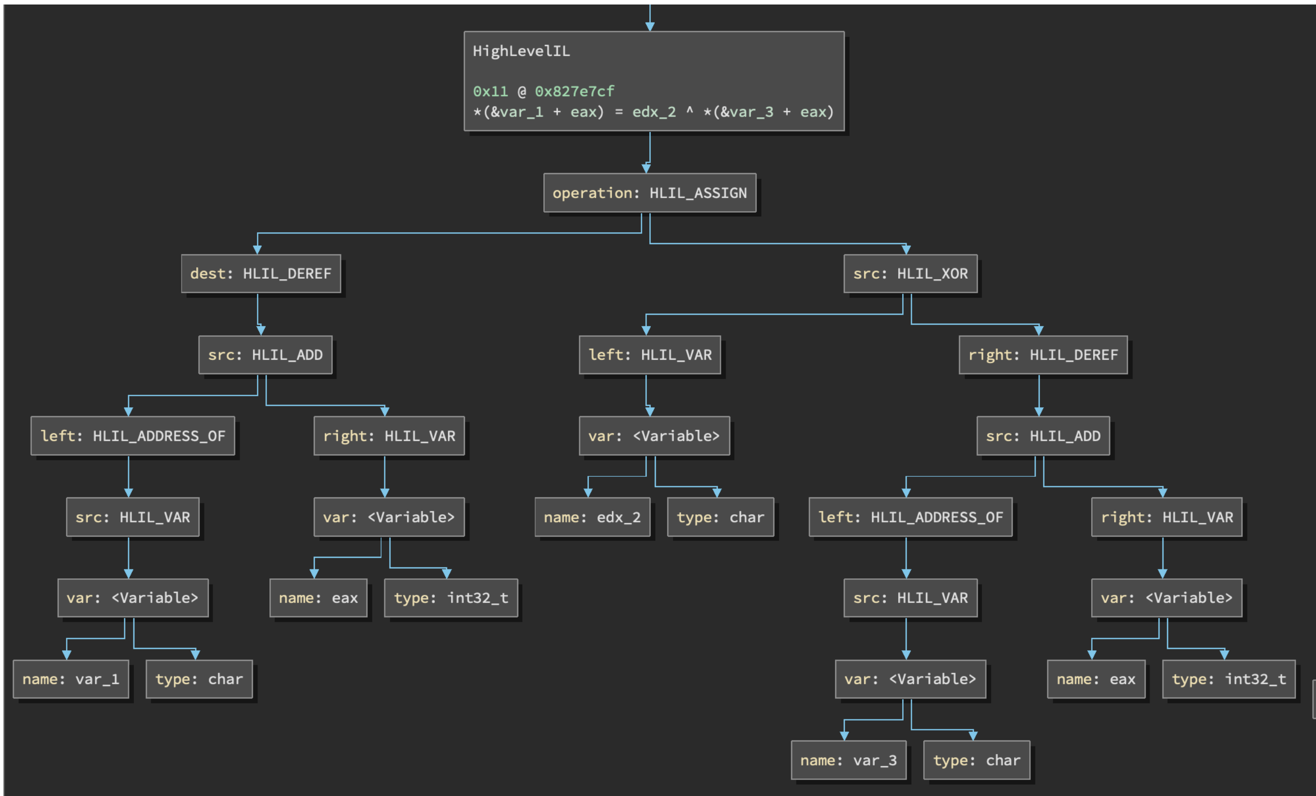
The solution will need to find functions in the binary which call `_runtime.slicebytetostring` and make use of an `xor` instruction.

In my first attempt you'll see that I'm not using the IL for this, but that's trivially solved in the final solution.

String Extraction - Attempt One

My first attempt at writing this didn't go very well… Binary Ninja implements multiple different ILs at varying levels of readability based on how much lifting, transformation and control flow recovery takes place. You can see examples of LLIL (low level intermediate language), MLIL (medium level intermediate language) and HLIL (high level intermediate language) ILs in the corresponding links.

So I made the assumption that HLIL would be the best choice because there was less text. I also didn't really understand that an IL operation could consist of multiple different IL operations:



Example of a XOR/assign operation in HLIL

You can see my first attempt here. I wish I had read Josh Watson's blog post on Binja's IL before I started working on it as it details some of the non obvious details 😅

I managed to get some results from this approach but it was not consistent and broke very easily.

String Extraction - Attempt Two

I started chatting to Jordan Wiens one of the founders of Vector35, the company who makes Binary Ninja – and he suggested that I should rewrite it using LLIL and a full blown emulator rather than a state machine.

At first this was daunting, but I started to play around with it and thanks to some incredibly helpful plugins which I'll detail later, as well as a project from Josh Watson which had the foundations of an LLIL emulator, I was able to start making headway.

Using a helpful snippet from Jordan I was able to work out what LLIL operations I needed to implement and in total there were around 28 of them:

IL operation	Description
`LLIL_TAILCALL`	Call another function without writing a return address to the stack
`LLIL_CALL`	Call another function and push a return address to the stack

IL operation	Description
LLIL_RET	Pop a return address from the stack and jump to it
LLIL_PUSH	Push a value onto the stack
LLIL_POP	Pop a value from the stack
LLIL_XOR	XOR 2 values
LLIL_ZX	Zero Extends
LLIL_GOTO	Set the instruction pointer
LLIL_STORE	Write some data to memory
LLIL_READ	Read some data from memory
LLIL_SET_FLAG	Sets a flag
LLIL_FLAG	Reads a flag
LLIL_CMP_NE	Is not equal
LLIL_CMP_E	Is equal
LLIL_CMP_SLE	Signed less than or equal
LLIL_CMP_SGT	Signed greater than
LLIL_CMP_SGE	Signed greater than or equal
LLIL_CMP_UGE	Unsigned greater than or equal
LLIL_CMP_UGT	Unsigned greater than
LLIL_CMP_ULT	Unsigned less than
LLIL_CMP_SLT	Signed less than comparison
LLIL_CMP_ULE	Unsigned less than or equal
LLIL_IF	Check a conditional and set the instruction pointer based on the outcome
LLIL_SET_REG	Set a register value
LLIL_CONST	Get a constant value
LLIL_CONST_PTR	Get a constant value that happens to be a pointer
LLIL_REG	Read a register value

IL operation	Description
`LLIL_SUB`	Subtract two values
`LLIL_ADD`	Add two values

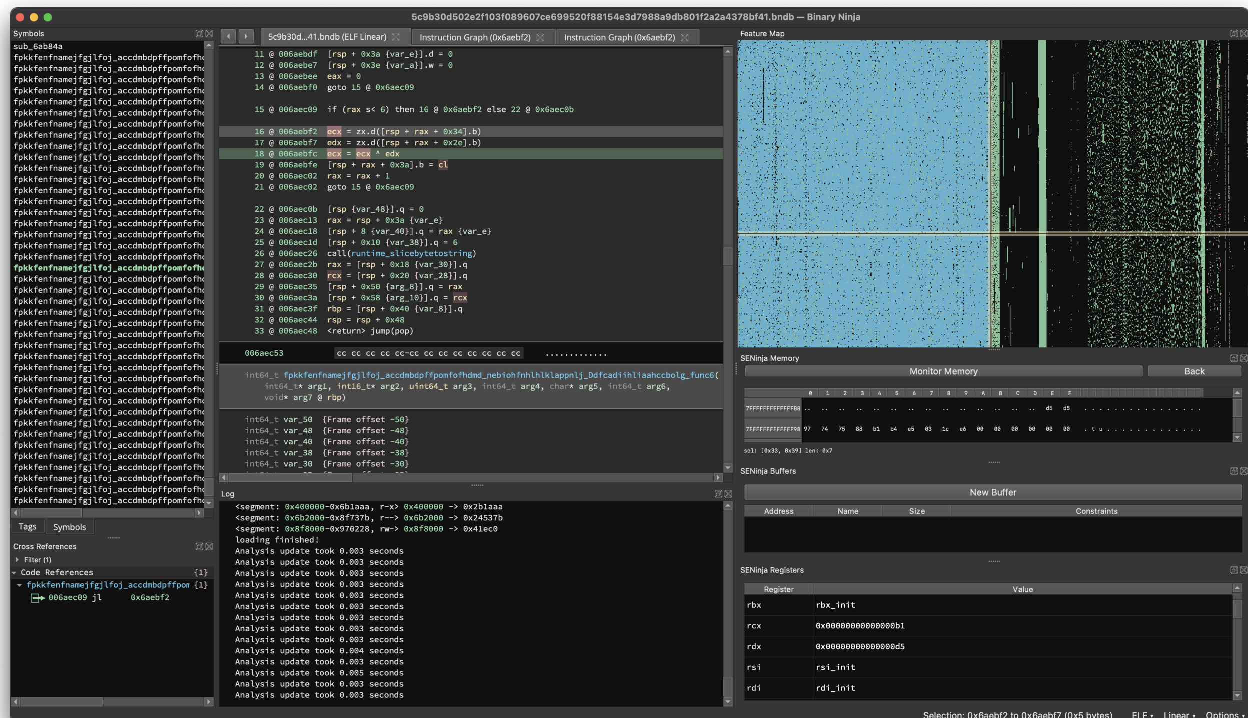
Alone none of these seem too complex right? But together they allow us to fully implement what is required to decode the XOR obfuscation in less than 500 lines of code. There's a huge amount of heavy lifting that Binary Ninja does under the hood that helps us keep the code as simple as possible, and I'm sure I'm missing some tricks.

Once I had something that *kinda* worked Jordan jumped in and converted it into a functional Binary Ninja plugin and cleaned up some of the code smell that occurs when you're hacking away on a problem.

While developing the plugin I came across some incredibly useful Binary Ninja plugins which I feel need a mention:

SENinja

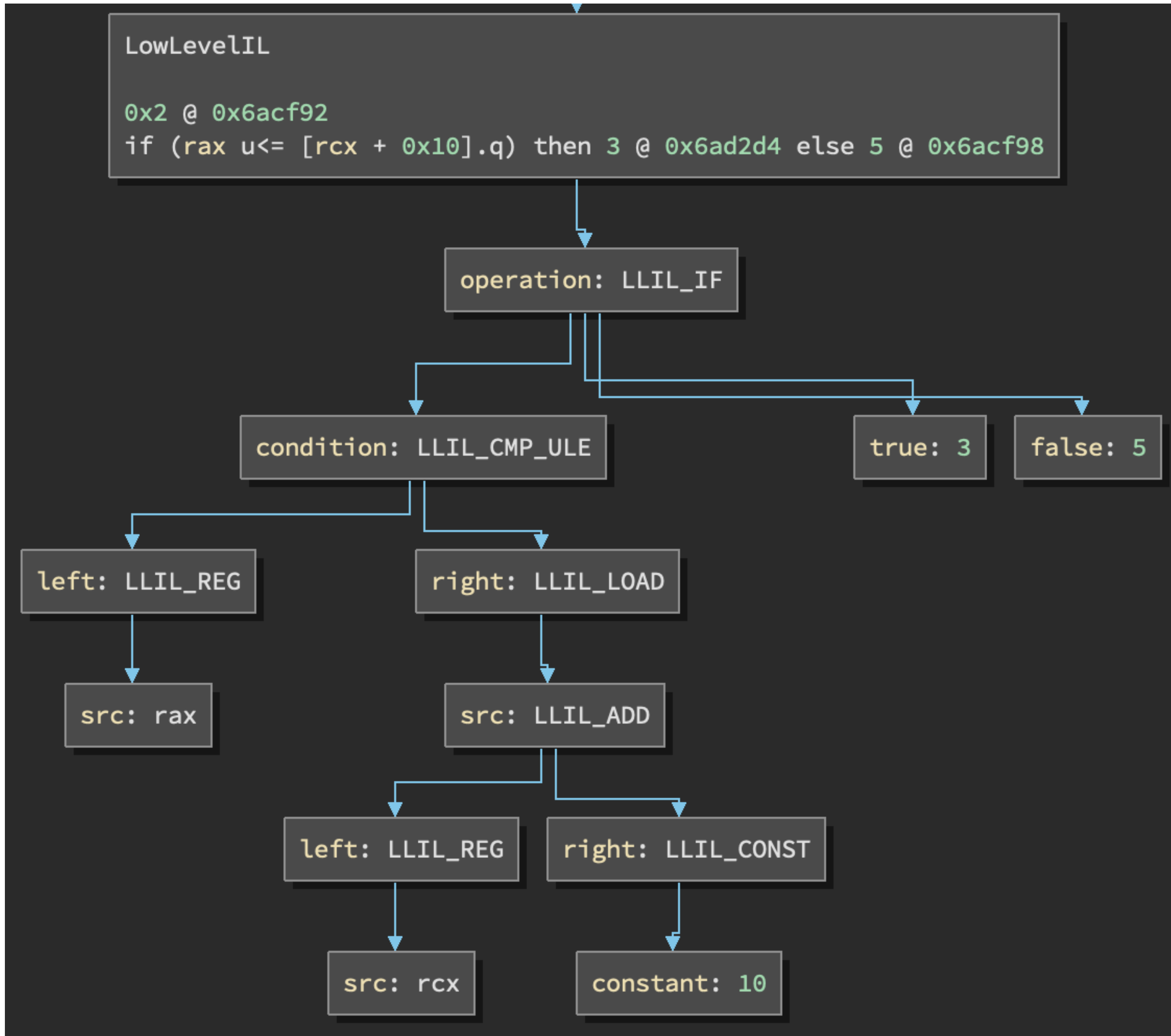
SENinja is a symbolic execution engine for Binja, built using the Z3 SMT solver. It implements a LLIL emulator that builds and manipulates Z3 formulae. Although the intentions of SENinja are much more complex then what I was using it for, it was really useful to have something I could compare with (I didn't want to turn on a Linux VM and use a debugger and deal with possible anti debugging tricks).



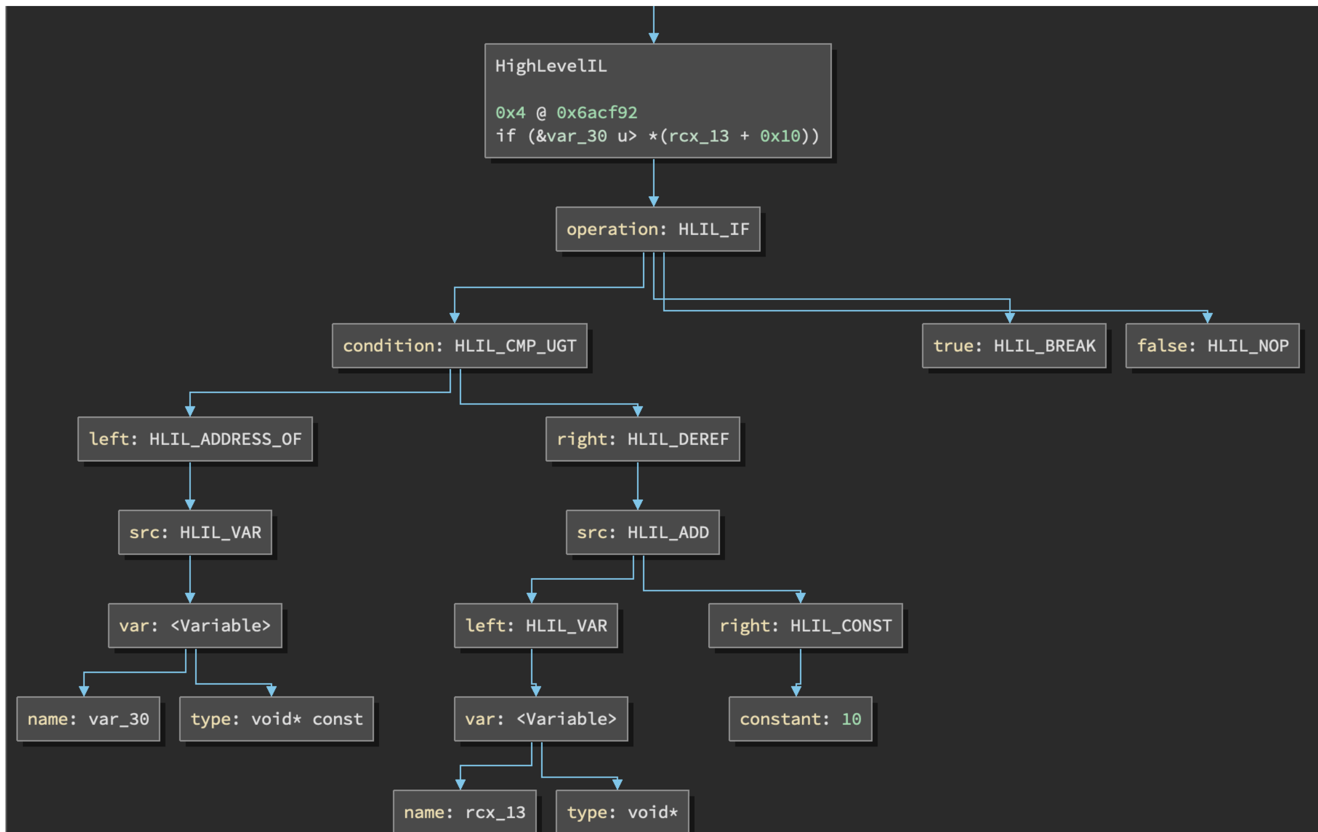
What SENinja looks like while emulating a function

BNIL Instruction Graph

BNIL Instruction Graph allows you to click on a line of IL and generate a graph of the IL operations that make up that line.



LLIL



HLIL

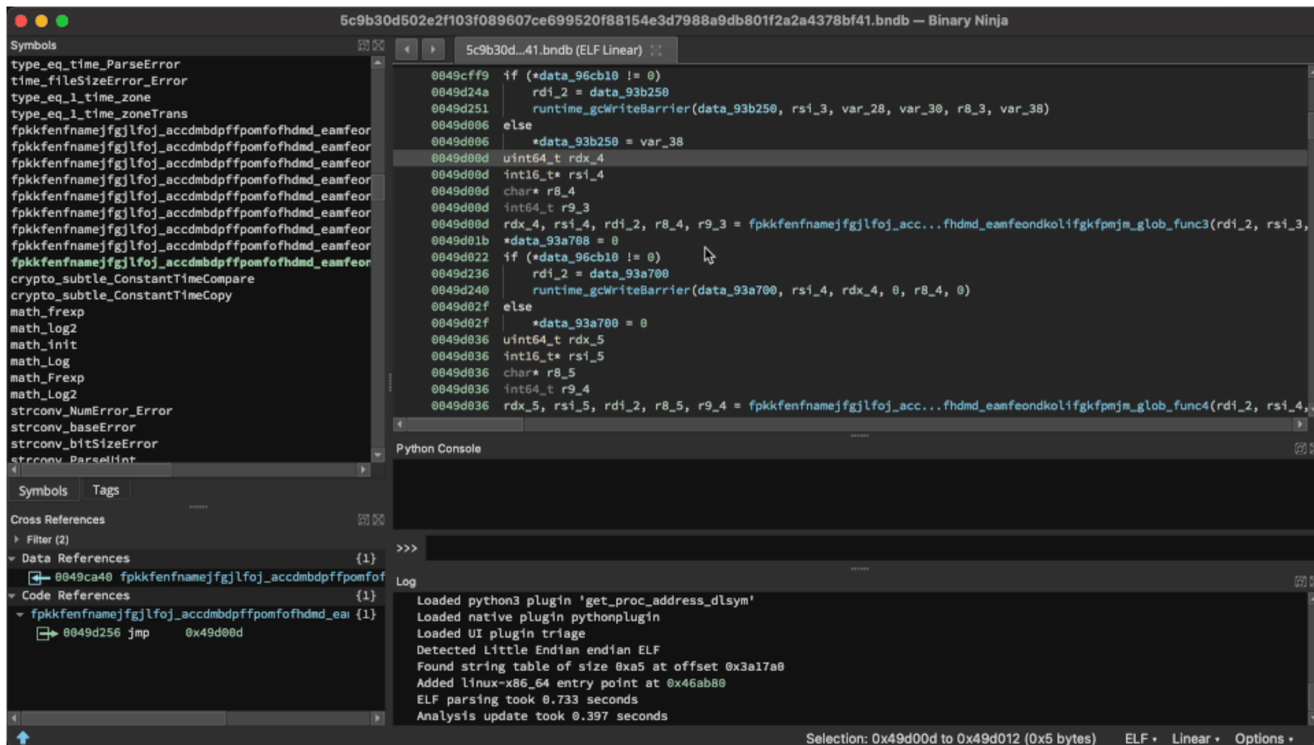
The difference in complexity between a line of LLIL and HLIL is surprising. However it makes sense once you understand that HLIL exists to allow higher level control flows to be recovered with the intention of producing a source code representation, typically in an emulator though you don't need this kind of high level control flow data.

Another powerful feature of BNIL Instruction Graph is the ability to generate specific IL matching templates that are convenient starting points for building IL code:

```
def match_LowLevelIL_10a6df6_0(insn):
    # ecx = ecx ^ edx
    if insn.operation != LowLevelILOperation.LLIL_SET_REG:
        return False

    if insn.dest.name != 'ecx':
        return False

    # ecx ^ edx
    if insn.src.operation != LowLevelILOperation.LLIL_XOR:
        return False

    # ecx
    if insn.src.left.operation != LowLevelILOperation.LLIL_REG:
        return False

    if insn.src.left.src.name != 'ecx':
        return False

    # edx
    if insn.src.right.operation != LowLevelILOperation.LLIL_REG:
        return False

    if insn.src.right.src.name != 'edx':
        return False

    return True
```

Can haz please?

As the usage of Golang by malware authors increases, so will their understanding and capabilities within the Go ecosystem. Right now gobfuscate is the main obfuscator used by actors using Golang, but this won't always be true. As a Hacker News comment puts it: this is fairly weak. Gobfuscator doesn't implement control flow obfuscation, the runtime functions aren't obfuscated and at most gobfuscate serves mostly as an annoyance and a thinly veiled layer of obscurity.



Solver at work!

You can download the Binary Ninja plugin on the plugin manager or download the code from the GitHub repository linked here. Additionally you can find the Blackrota 32-bit and 64-bit Binary Ninja databases to look at on Binary Ninja cloud.

Thanks:

- A massive thanks to the people in the Binary Ninja Slack who answered some of my questions while I was getting started.
- Josh Watson, for his permissibly licensed emulator that I was able to use to get an idea of what was required to achieve my goal.
- Jordan Wiens, for providing assistance during development, cleaning up messy code, filing a couple of Binja bug reports for me and generally answering questions about Binary Ninja.