Ginko: Obfuscation Bypass

Gum Stalker for behavioural analysis

Introduction

Obfuscation is a common and relatively simple way to deter curious minds. Common in commercial applications such as paid tools, banking apps, and social media apps (Snapchat), this technique makes static analysis challenging. Tools like Binary Ninja and IDA Pro are often able to sift out pseudocode and high level IR correctly, but certain patterns make control flow untraceable. The most simple example is indirect control flow. With this technique, direct branches are replaced with branches to registers, which can be further obfuscated by storing the destination in the stack or XOR encrypting, or pretty much anything you can think of. A creative obfuscator will combine techniques like replacing simple instructions with more obscure and unpredictable patterns. Control flow flattening is a popular technique as well, used in LLVM-Obfuscator. A graph view of a function would show bogus codepaths that the executable never attempts. Commercial and private obfuscators even alter the ABI to pass arguments to functions in strange ways, and do indirect branches to other parts of the code using those arguments.

I can think of a million ways to really get under the skin of anyone doing static analysis. It would be somewhat trivial to encrypt a jump table until runtime, which is well beyond the capabilities of static analysis and symbolic execution.

What About GDB

The issue with just using a debugger is that processes can usually tell they are instrumented, and single stepping usually literally patches and unpatches the next instruction with a trap to catch the control again. Hardened executables can even trick debuggers by creating branch patterns that cause the program to spinlock. All around, not super useful for opaque executables and quickly scanning the control flow.

Stalking

The Frida project is really incredible. It offers support for virtually every architecture for its tools, which include Gum Stalker. This tool allows for blocks of code to be reinstrumented. What this means is that by copying a block of code into a different code space, it is possible to patch out instructions that read the program counter (that would indicate that the code is instrumented) and still maintain execution of the process undisturbed. The developers even went so far as to handle instructions that set flags, like AARCH64's cmp.

Typically this is done through a javascript interface and the frida-gadget interface, a binary that is loaded alongside the target.

Ginko

I made a quick tool called Ginko. It uses the Gum Stalker devkit available from Frida to statically link the stalker tool, and uses the C bindings to begin instrumentation. The goal is to create a log of the control flow that can be used alongside static analysis to make sense of bogus control flow and encoded jumps. Using the Capstone disassembler, it is easily extensible for functionalities like printing the result of obscure ALU instructions.

So let's go. We are targeting XNU platforms (iOS) where dynamic libraries get to execute initializers when loaded. In our constructor we select the thread we are executing on and register a transform block as a callback to process events.

```
__attribute__((constructor)) static void stalker_init(void) {
    if (!open_log_file()) return;

    gum_init_embedded();

    g_ctx.main_start = 0;
    g_ctx.main_end = 0;
    gum_process_enumerate_modules(on_module, &g_ctx);

    if (g_ctx.main_start == 0) {
        fprintf (log_file, "[STALKER] ERROR: could not locate module
named 'main'.\nEnsure the executable's basename is 'main'.\n");
        return;
    }

    fprintf (log_file, "[STALKER] Main module: 0x%012" G_GINT64_MODIFIER
"x" " - 0x%012" G_GINT64_MODIFIER "x\n", (guint64) g_ctx.main_start,
(guint64) g_ctx.main_end);

    g_ctx.main_thread_id = gum_process_get_current_thread_id();
    fprintf(log_file,"[STALKER] Main thread id: %u\n", (guint)
g_ctx.main_thread_id);

    g_ctx.stalker = gum_stalker_new();
```

```
    gum_stalker_set_trust_threshold(g_ctx.stalker, -1);

    //gum_stalker_activate_experimental_unwind_support();

    g_ctx.transformer =
gum_stalker_transformer_make_from_callback(transform_block, NULL, NULL);

    gum_stalker_follow(g_ctx.stalker, g_ctx.main_thread_id,
g_ctx.transformer, NULL);

    fprintf(log_file, "[STALKER] Attached to main thread. Logging
begins.\n");
    fflush(log_file);
}
```

The callback handles each block, and we can choose to monitor each instruction passing through. A simple check is performed to determine if the instructions belongs to the target binary, or some function it is calling from the shared cache. It is not really important here to understand what instructions are being executed by strcmp for example, so no instrumentation is performed out of the main binary.

```
static void transform_block(GumStalkerIterator *iterator,
GumStalkerOutput *output, gpointer user_data){
    // This is called for every basic block that the Stalker compiles
    const cs_insn *insn;

    while (gum_stalker_iterator_next (iterator, &insn)) {
        gum_stalker_iterator_keep (iterator);
        if (insn->address >= g_ctx.main_start && insn->address <
g_ctx.main_end)
            gum_stalker_iterator_put_callout (iterator, per_instruction,
(gpointer)insn, NULL);
    }
}
```

Okay now how do we handle each instruction? Gum Stalker includes a version of Capstone that can generate mnemonics for any instruction,

which we can leverage to print the code path to a log file. Additionally, a structure is provided that shows the entire CPU context for that instruction's execution.

```
static void per_instruction(GumCpuContext *cpu_context, gpointer
user_data){

    cs_insn *insn = user_data;

    if (insn->address >= g_ctx.main_start && insn->address <
g_ctx.main_end) {
    g_insts++;
        fprintf(log_file, "[STALKER][%-14p] %-8s %-12s\n", (void
*)insn->address, insn->mnemonic, insn->op_str);
        fflush(log_file);
    } else if (SHOULD_MESS_AROUND){
        Dl_info info;
        char *module_name = "unknown";
        if (dladdr((void *)insn->address, &info) && info.dli_sname) {
            module_name = info.dli_sname;
        }
        fprintf(log_file, "[STALKER][%-14p] %-8s %-12s [%s]\n", (void
*)insn->address, insn->mnemonic, insn->op_str, module_name);
        fflush(log_file);

    }

    fflush(log_file);
}
```

In the primary case, this function serves to double check the address of the execution, log the address and instruction, and count the instructions. SHOULD_WE_MESS_AROUND? Probably not? That functionality not only logs every instruction but also does a dladdr lookup to determine which module is being executed. For example, if the main binary is being stalked and calls NSGetExecutablePath, the dladdr would report that Foundation binary is being executed. These functions are less interesting to read through because they include provisions for pointer authentication, multithreading, and other safety things that add a lot of bloat we don't want to sort through.

So what do the results look like? Here's a snippet from a test on an x86-64 MacOS build. Blocks that have been already logged are automatically omitted by stalker, which decreases the log size significantly. This gives a pretty clear reference for static analysis.

```
[STALKER][0x10dd379fc   ] call     0x10dd37ba0
[STALKER][0x10dd379e8   ] mov      al, 0
[STALKER][0x10dd379ea   ] call     0x10dd37ba0
[STALKER][0x10dd37b49   ] lea      rdi, [rip - 0x130]
[STALKER][0x10dd37b50   ] call     0x10dd37990
[STALKER][0x10dd37b49   ] lea      rdi, [rip - 0x130]
[STALKER][0x10dd37b49   ] lea      rdi, [rip - 0x130]
[STALKER][0x10dd37b50   ] call     0x10dd37990
[STALKER][0x10dd37a19   ] ret
```

Another test on jailbroken iOS against a heavily obfuscated binary:

```
[STALKER] Main module: 0x000100ad8000 - 0x000100ae0000
[STALKER] Main thread id: 259
[STALKER] Attached to main thread. Logging begins.
[STALKER][0x100adc424   ] sub      sp, sp, #0x70
[STALKER][0x100adc428   ] stp      x29, x30, [sp, #0x60]
[STALKER][0x100adc42c   ] add      x29, sp, #0x60
[STALKER][0x100adc430   ] stur     wzr, [x29, #-4]
[STALKER][0x100adc434   ] adrp     x0, #0x100adc000
[STALKER][0x100adc438   ] add      x0, x0, #0x905
[STALKER][0x100adc600   ] adrp     x16, #0x100ae0000
[STALKER][0x100adc604   ] ldr      x16, [x16, #0x58]
[STALKER][0x100adc440   ] adrp     x0, #0x100adc000
[STALKER][0x100adc444   ] add      x0, x0, #0x923
[STALKER][0x100adc440   ] adrp     x0, #0x100adc000
[STALKER][0x100adc444   ] add      x0, x0, #0x923
[STALKER][0x100adc44c   ] stur     wzr, [x29, #-8]
[STALKER][0x100adc454   ] ldur     w8, [x29, #-8]
```

Bugs

The test on AARCH64 reveals some issues. For some reason, branch calls are omitted. This is likely because the address of the instruction (which we choose to not log if it is outside the main binary) is the value of the program counter AFTER the instruction executes. So out of binary branches are simply not logged. The fix? A simple static variable to hold the previous address of execution to use in comparisons. Another frustrating quirk of gum_stalker is that blocks that have already been processed are not reprocessed. Stalker simply considers them “safe” to execute in the original code page and then doesn't bother to copy them

over and do all its magic every time. This doesn't really affect the introspective value, in Ghidra/IDA Pro/Binary Ninja it is simple to just make a note of the function or give it a description. Oftentimes it is just a simple memcpy implementation.

Findings

The most effective way to track the control flow of a program without modifying the code is to copy it somewhere else, handle edge cases, and execute it there.

Jailbreak detection in heavily obfuscated binaries now becomes less opaque, because it is trivial to see that the process may call out to vm_reigon_check64, compare the result (in between indirect branches and bogus dead end codepaths) and store it somewhere using a convoluted process. Rather than sifting through the static binary, the attacker can leverage Ginko/Stalker to see where it writes to, what it calls, which path it takes, which comparisons actually matter, A/B test different bypass methods, the list goes on.

Next Steps

Stalker makes this functionality extremely portable, but introduces complexity, overhead, and unwanted optimizations and features. I have a few ideas how to improve this.

Full symbolic execution on bare metal. This means reading the target binary instruction, logging it, jumping to some part of our code where we have a copy of that instruction (for certain instructions that read PC or jump, this would handle edge cases where we need to lie to the program we are symbolically executing), and so on. Performance-wise, a terrible idea. It would be more like emulating the entire program, but using the real hardware to execute it. It would offer insane introspection but absolutely no portability. It would also have the advantage of not requiring page permissions to be switched from write to execute and vice versa, but would require every combination of operations and registers to be in the tool’s __text.

Another option is remaking stalker to target a specific arch and logging everything. So copy the victim’s code to a new page, do the patches needed to maintain control and invisibility, execute it, return to our handler, log stuff, and then load another block.

Last option: design a CPU with the same architecture and hardware debugging support, do layout, tapeout, and fabrication at scale with a

9nm process and trillions of transistors, and place it in a test device. It would only cost several million dollars and a lifetime of work…

Alright thanks for reading my crude security research