GitHub Bug Used to Infect Game Hackers With Lua Malware

Published: 2024-03-03 · Archived: 2026-04-05 15:45:45 UTC

Overview

Malware operators are using a cloned game cheat website, SEO poisoning, and a bug in GitHub to trick would-be-game-hackers into running Lua malware. Our notes are divided into two sections, the first part is focuses on the GitHub bug the enables the malware delivery and the second part covers our attempt to analyze the Lua JIT malware.

Special Thanks

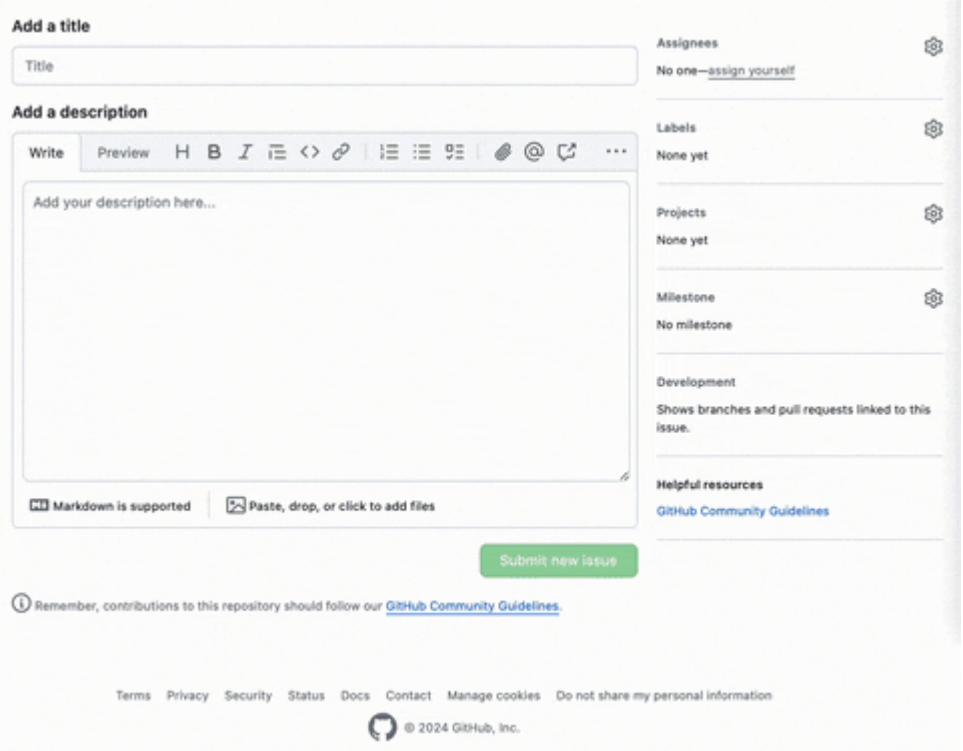
- @JustasMasiulis
- 0AVX
- Fish-Sticks
- Jollyc
- @xusheng6
- *Themida* for the sample 😉

Delivery

The game cheat that has been targeted is an open source aim bot called AIMMY which is maintained in the following GitHub repository Babyhamsta/Aimmy with the website aimmy.dev.

The malware operators have made a clone of the GitHub repository here `https[:]//github[.]com/nehuenbohm/Aimmy` and setup a malicious website `https[:]//aimmy[.]app` which links to the cloned GitHub repository. The website also has a modified download link which **links to malware** hosted at `https[:]//github[.]com/Babyhamsta/Aimmy/files/14475029/Aimmy.zip`. Note that this download link is for a ZIP file that is **hosted on the original benign GitHub repository**.

The malicious website has a high Google rank (likely due to SEO poisoning) and is ranked above the legitimate aim bot website. Users searching for the aim bot are likely to be tricked into visiting the malicious site and downloading the malware.



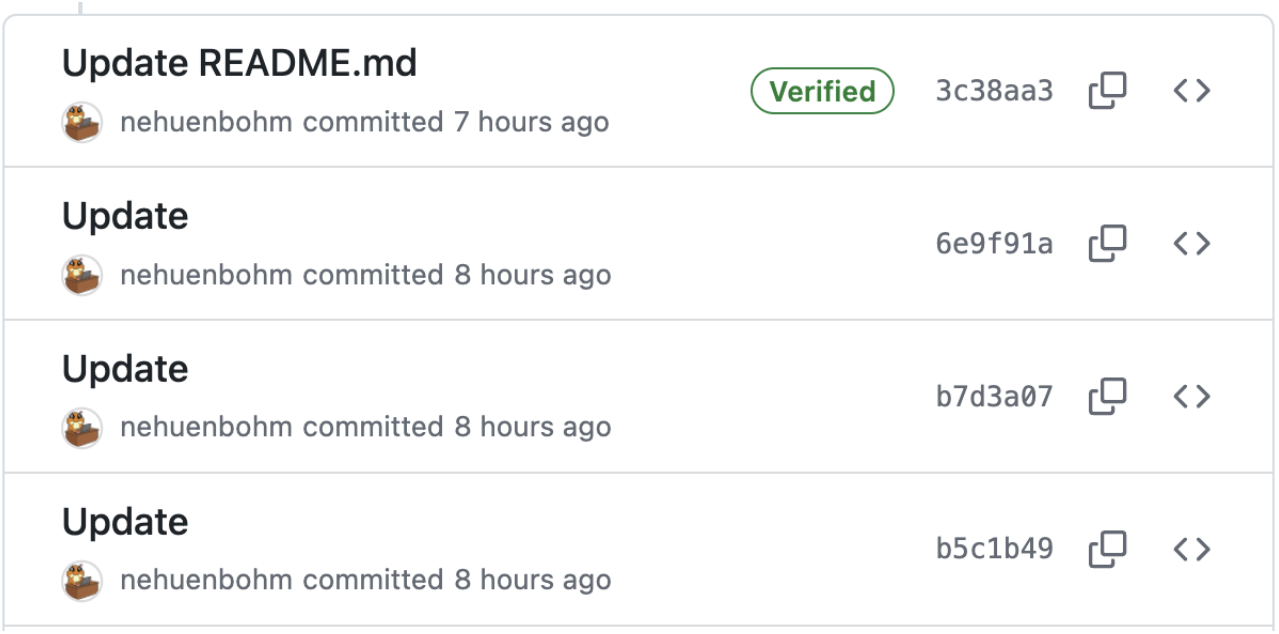
A key element in the attack involves hosting that malware payload on the original benign repository. This both distances the operators from the payload host and also may enable the download URL to blend in with normal traffic. Using this trick makes it difficult for the victim to identify that they are clicking a malicious link. But how are the operators able to host their malware on another GitHub repository? This had us stumped until @JustasMasiulis discovered the bug … feature?

When opening an issue on a repository **any file uploaded to the issue is stored under the GitHub repository where the issue when opened**. These files persist even if the issue is never saved. This means that anyone can upload a file to any git repository on GitHub an not leave any trace that the file exists except for the direct link.

- Cick `New Issue` on the target GitHub repository
- Drag and drop malware into issue description
- Copy generated link for malware from description
- Close issue without submitting/saving
- The link to the uploaded malware remains active even though the issue was never saved

@JustasMasiulis posted a POC of this to X last night.

Malicous GitHub Cloning



The malware operators also used a clever technique to obscure the fact that they have cloned the original GitHub repository. ArsTechnica recently detailed similar malicious cloning attacks but in this case we were able to observe the attack in real time, and can report some additional details.

- The target repository is downloaded then uploaded to a new account, it is not forked.
- Though the git commit history is maintained in the newly uploaded repository it does not display as a fork of the original repository.
- The operator then makes **thousands of empty commits** to the repository, likely scripted, making it appear as though they are main contributor to the project. In the case of the aim bot the operator made over seven thousand commits in a 24h hour period.
- The final commit is made to the project README changing the project download links to the malware URL. This commit is made as a GitHub verified user as the user is verified for the repo.

Lua Malware Analysis

The malware itself is very unique. It consists of a Lua JIT file, a compiled LuaJIT executable used to interpret the JIT file, and a batch script used to elevate permissions and run the JIT in the interpreter.

Sample

The malware is delivered in a ZIP file malshare

- c912762952152c40646a61d7cc80a74f61ddd7aad292a1812f66e76b405f9660 `Aimmy.bat`
  - Batch script used to run the lua code in the interpreter
- 1cf20b8449ea84c684822a5e8ab3672213072db8267061537d1ce4ec2c30c42a `AimmyLauncher.exe`
  - LuaJIT intepreter
- d6d3c8ea51a025b3abeb70c9c8a17ac46cf82e5a46a259e9aaa9d245212d9e17 `README.txt`
- fa3224ec83c69883519941c0e010697bcdc0518a3d9e2c081cd54f5e9458b253 `data`
  - Malicious compiled Lua JIT code, magic bytes `1B 4C 4A`

- ff976f6e965e3793e278fa9bf5e80b9b226a0b3932b9da764bffc8e41e6cdb60 `lua51.dll`

Analysis

Using the LuaJIT Decompiler v2 we were able to decompile the JIT and recover the Lua code only find that it had been heavily obfuscated.

```
local var_0_19 = {
    var_0_17[var_0_18("\xA7Z$\\o\x19.", 8170536974433)],
    var_0_17[var_0_18("\x83\xEB\xC1\xEC\xAER\x0E\xCB", 32340223605166)],
    var_0_17[var_0_18("\x98E\r\f", 18847752807337)],
    var_0_17[var_0_18("Z\x16g\xD2", 17704612011450)],
    var_0_17[var_0_18("K\x18^\xAE\xF8_\xFF", 2538651564100)],
    var_0_17[var_0_18("\x1Bdo\xF6B\xF3\x92\xC0\x84^zg\x8B\xB1\x95\xC3", 17854945091482)],
    var_0_17[var_0_18("\xFFYk\xF5\xAB37\x83", 21489022010759)],
    var_0_17[var_0_18("\x8Eq\x85\x93yJ.\xD8", 17231694304248)],
    var_0_17[var_0_18("\xDD,G\x17\xEBg#h", 1115184245546)],
    var_0_17[var_0_18("DE\x10D", 12159446557750)],
    var_0_17[var_0_18("(\x92\xE7\xB9X\xEE8C\x1F\x19Z\xBE\x8Fq\xBF", 24217461828912)],
    var_0_17[var_0_18("\xF8cQsU\xAF\x9C\x9C", 12518619714798)],
    var_0_17[var_0_18("o\x01\xC1\xC7\xE7$(\x84", 29604450893836)],

 . . .
```

The obfuscation was matched to an open source Lua obfuscator called Prometheus. This obfuscator bot encrypts strings, and employs a virtual machine to protect the Lua code.

Lua Environment Instrumentation

Rather than attempting to break the VM directly we attempted to trace the Lua code dynamically. Because the malicious Lua code is passed as an argument to the LuaJIT interpreter we first attempted to instrument the Lua environment by loading tracing code prior to running the JIT, a concept based on the ideas outlined in Nick's blog Hooking LuaJIT. Using the following script passed with the `-e` argument to the interpreter.

```
 AimmyLauncher.exe -e <script> data
```

```
jit.off()

FILE_PATH = "C:\\LuaJitHookLogs\\"
STARTING_TIME = os.clock()
GDUMPED = false

function table.show(t)
```

```
    local function serialize(arr, level)
        local str = ""
        local indent = string.rep(" ", level*2)

        for i, v in pairs(arr) do
            if type(v) == "table" then
                str = str .. indent .. i .. ":\n" .. serialize(v, level+1)
            else
                str = str .. indent .. i .. ": " .. tostring(v) .. "\n"
            end
        end

        return str
    end

    return serialize(t, 0)
end

function dumpGlobals()
    local fname = FILE_PATH .. "globals_" .. STARTING_TIME .. ".txt"
    local globalsFile = io.open(fname, "w")
    globalsFile:write(table.show(_G, "_G"))
    globalsFile:flush()
    globalsFile:close()
end

function trace(event, line)
    local info = debug.getinfo(2)

    if not info then return end
    if not info.name then return end
    if string.len(info.name) <= 1 then return end

    if (not GDUMPED) then
        dumpGlobals()
        GDUMPED = true
    end

    local fname = FILE_PATH .. "trace_" .. STARTING_TIME .. ".txt"
    local traceFile = io.open(fname, "a")
    traceFile:write(info.name .. "()\n")

    local a = 1
    while true do
        local name, value = debug.getlocal(2, a)
        if not name then break end
        if not value then break end
```

```
        traceFile:write(tostring(name) .. ": " .. tostring(value) .. "\n")
        a = a + 1
    end

    traceFile:flush()
    traceFile:close()
end
debug.sethook(trace, "c")
```

This had some limited success, there is an anti-tamper feature in the VM (which was eventually bypassed by Jollyc) but I ran into some issues... mainly I don't know anything about Lua and troubleshooting the errors was time consuming.

Lua Garbage - Dumping Encrypted Strings

This idea belongs to **Jolyc**, **Fishy-Sticks**, and **0AVX**. As Fishy-Sticks put it...

> Lua is a managed language which has a garbage collector, me and AVX are looking for where they collect strings so we can dump all string info before it gets collected thus holding decrypted strings :) LuaJIT is a little bit diff from normal Lua so it takes us a bit to find All that was needed was to clone the LuaJIT repository, locate the garbage collector for the strings in the LuaJIT code, and add our own hook to dump them before they were freed. We could then compile our custom version of LuaJIT and use this to run the malicious Lua code and dump the strings.

Hook lj_str_free

The `lj_str_free` function in `lj_str.c` is responsible for freeing the string memory. By adding some simple log code here and recompiling

```
void LJ_FASTCALL lj_str_free(global_State *g, GCstr *s)
{
    const char* myStr = strdata(s);
    if (myStr) {
        printf("STRING: %s\n", myStr);
        FILE* newfile = fopen("log.txt", "a");
        fprintf(newfile, "STRING: %s\n", myStr);
        fclose(newfile);
    }
  g->str.num--;
  lj_mem_free(g, s, lj_str_size(s->len));
}
```

Additional hooks can be added in `lib_os.c` on the os functions to dump interactions between the Lua and the host.

```
LJLIB_CF(os_execute)
{
    const char* cmd2 = luaL_optstring(L, 1, NULL);
    if (cmd2) {
        printf("OS_EXECUTE: %s\n", cmd2);
        FILE* newfile = fopen("log.txt", "a");
        fprintf(newfile, "OS_EXECUTE: %s\n", cmd2);
        fclose(newfile);
    }
```

Because many string operators result in partial strings being allocated to memory there is a lot of noise in the dump but we were able to recover full structs, function definition, as well as strings used for the C2 communication and persistence of the malware.

TODO - this approach could be combined with more classic API tracing to provide an more holistic view of the malware.
