Toward Partial Proofs of Vulnerabilities

Jonathan Brossard

Toward Partial Proofs of Vulnerabilities

Jonathan Brossard
dept. CEDRIC
Conservatoire National des Arts et Métiers
Paris, FRANCE
jonathan.brossard@lecnam.net

***Abstract*—With the adoption of compelling legislation regulating Product Security, quickly replicating or disproving the presence of publicly known vulnerabilities is becoming an essential part of the software life cycle. Moreover, third-party code is often integrated and shipped in binary form rather than source code. In this article, we present an original debugger named the Witchcraft Shell (WSH), aiming at helping software maintainers validate the existence of public vulnerabilities when source code is unavailable. By making C/C++ dynamically linked ELF executables scriptable and their internal functions callable with no or little context, we open the way to new heuristics: "partial proofs of vulnerabilities". This tool is published under a permissive dual BSD/MIT open-source license.**

***Index Terms*—exploitability, decidability, libification, procedural debugging, proof of vulnerability**

I. Introduction

The Executive Order on Improving the Nation's Cybersecurity [2] in the US and the Cyber Resilience Act [10] [8] in Europe render the creation and publication of Software Bills of Materials [9] [27] (SBOMs) to customers compulsory. Since they contain exact versions of all the software components, SBOMs make version-based public vulnerability assessments more transparent.

However, ensuring that a potential CVE actually affects a piece of software typically involves writing an exploit for this software [24] or, at the very least, triggering the vulnerability. Unfortunately, even this first step is an undecidable problem in general [14]. This creates a situation where Product Security teams must solve undecidable problems repeatedly. Because existing data suggest they make up to 70% of exploitable vulnerabilities in C/C++ applications [15], we will focus on memory corruption vulnerabilities [18] in the remainder of this article.

To help tackle this issue, we offer a methodology to evaluate the existence of CVEs in compiled ELF executables. Building on our previous research [3] [4], we present an original debugging technique that transforms executables into scriptable programs and allows calling vulnerable functions directly and without context, hereby avoiding the reachability problem entirely when attempting to prove or disprove the presence of CVEs within an application. We name this heuristic, our main contribution, "partial proofs of vulnerabilities".

To illustrate the benefits of this methodology, we publish two WSH scripts that may be used to assess the presence of CVE-2022-3602 and CVE-2022-3786 in compiled versions of OpenSSL, without relying on source code.

II. Previous Work

Fundamental control-flow techniques such as disassembly [26] or decompilation [25] are undecidable in general. However, Duck et al. have managed to perform local modifications of executables without control-flow recovery [13]; Shapiro et al. by modifying only ELF metadata. [22]

To perform an intraprocedural analysis of a CVE within a given function without facing the reachability problem [21], it would be convenient to perform a simpler and deterministic preliminary transformation of the binary to be analyzed into a shared library. This way, its internal (non-static) functions would be exposed and callable. Transforming static ELF libraries into shared libraries, for instance, may be done using the –whole-archive option of the GNU linker [23].

Capeletti [7] has described how to transform ELF executables into object files, hence performing an "unlinking" operation (undoing the work of a linker). Previous research has also shown that loading Microsoft Windows libraries within Linux executables [20] or transforming Windows Portable Executables (PE) into (DLL) shared libraries [12] is practical.

We have previously published [3] a tool to "libify" ELF [11] binaries, id est: to transform dynamically linked ELF binaries into shared libraries by modifying their metadata [4].

The steps to perform a libification are the following: First of all, modify the e_type member of the ELF header to match ET_DYN since all shared libraries match this type (as opposed to ET_EXEC). Then, parse the .dynamic section and replace DT_BIND_NOW entries with DT_NULL. Remove the bits DF_1_PIE and DF_1_NOOPEN in DT_FLAGS_1 if present. Finally, optionally set DT_INIT_ARRAYSZ and DT_INIT_ARRAY, DT_FINI_ARRAYSZ and DT_FINI_ARRAY in the .dynamic section to zero, to prevent constructors and destructors from being invoked at load time respectively.

This "libification" is realized by the wld tool (Witchcraft Linker) and then allows loading arbitrary dynamically linked ELF executables using the dlopen() function of the dynamic linker [1]. The address of arbitrary functions loaded this way in memory may be obtained via the dlsym() function of the dynamic linker.

Now that the principle of transforming ELF binaries into shared libraries that may be loaded in the address space of other applications is established, let us discuss how to design a more complex tool, the Witchcraft Shell (WSH), to help study the exploitability of vulnerabilities in those binaries.

III. The Witchcraft Shell: Design of a Procedural Debugger

A. Libification-Based Debugging

The Witchcraft Shell is a simple debugger, loading libified binaries into its own address space via dlopen(). This defers the intricate problems of recursively loading dependencies and solving relocations to the dynamic linker, rendering debugging simpler, less error-prone and ultimately more robust. This also simplifies memory management: since the debugger and the debuggee share the same address space, accessing the debuggee's memory does not require any system call, such as ptrace(), nor any other Inter Process Communication (IPC).

B. Functions Enumeration without Control-Flow Analysis

The dynamic linker features a convenient way to traverse all the binaries and libraries mapped in the address space of a process via a linked list of structures named link_map. This allows WSH to recursively find all the symbols [11], including their types, scopes, and names, exported by all the ELF files mapped in memory. The memory addresses of every function can then be found via calls to dlsym().

C. Arbitrary, Contextless, Function Calls

The calling convention of Linux C/C++ x86-64 userland applications is documented in the System V application binary interface and its AMD64 architecture processor supplement [19]. The stack frame, parameters passing (via general registers for the most part[1]), and order of arguments being known, it becomes possible to create a minimal "context" (stack frame) to call an arbitrary function in memory by transferring execution to the address returned by dlsym() for the desired function.

It is worth noticing that setting extra arguments, corresponding to extra registers, has no impact on execution. As such, knowing the exact number of arguments to a function is, in fact, not mandatory when performing a function call. Argument types also do not matter, general registers may be understood as untyped 64bits quantities (akin to a void* for higher level C/C++ developers) in all generality. Examining the return value of this function is then performed by reading the content of register RAX.

D. Scripting Binaries with the Witchcraft Shell

To enable interactive debugging, a lightweight Lua [17] interpreter has been embedded within WSH. Lua has been chosen for of its minimal footprint while offering a fully object-oriented, functional programming language under a permissive MIT license. Every function discovered within the address space is made visible to the Lua interpreter. They may then be called with any number of arguments, directly from Lua.

Users may subsequently write Lua scripts within WSH, but also invoke any function within the address space directly from their scripts or the Lua prompt. As a result, ELF executables loaded within WSH become entirely scriptable.

[1]Via in order RDI, RSI, RDX, RCX, R8, R9 for integers or pointers, and XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 for floating-point arguments.

E. Avoiding Reachability Problems via Procedural Debugging

Finally, the Witchcraft Shell may perform calls to arbitrary functions in isolation, without having to craft an input from the entry point of the debugged program, hence liberating the analysis from the reachability problem. We name this technique "procedural debugging".

IV. Partial Proofs of Vulnerabilities

The most critical factor determined by Guo et al. [16] as predicting the ease of fixing a vulnerability is the ability of developers to replicate it, hence understanding its root cause and being able to fully test fixes. As such, "partial proofs of vulnerabilities" in the form of WSH scripts allowing developers to trigger bugs directly, experiment with the vulnerable code, and use WSH scripts as unit tests for patches could help fix bugs more efficiently.

Exploiting a memory corruption vulnerability may be split into three stages: reaching the vulnerable function, triggering the vulnerability, achieving arbitrary code execution [24].

The Witchcraft Shell may be seen as a tool to help Product Security teams focus solely on triggering vulnerabilities.

It should be outlined, however, that a vulnerability may be triggerable via WSH, but not in fact reachable. For instance, if the vulnerable statement pertains to a function not connected to the application call graph, such as dead code, then triggering the vulnerability via WSH does not constitute sufficient proof of vulnerability.

As such, we name the heuristic of being able to trigger a vulnerability via WSH a "partial proof of vulnerability" in lieu of a definitive one.

V. Evaluation

Appendixes VIII.A and VIII.B contain WSH scripts that may be used to test if a given compiled software is vulnerable to CVE-2022-3602 and CVE-2022-3786 respectively. A stable DOI [6] has been created with a Dockerfile and instructions to replicate evaluation against multiple versions of OpenSSL with ease.

VI. Conclusion

In this article, we introduced the Witchcraft Shell (WSH), a procedural debugger leveraging the libification of executables. By making executable binaries scriptable, WSH scripts open the way to "partial proofs of vulnerabilities", heuristics able to trigger vulnerabilities despite the reachability problem.

VII. Availability

The Witchcraft Shell (WSH) detailed in this article is part of the Witchcraft Compiler Collection (WCC) reverse engineering framework [5]. It is available from https://github.com/endrazine/wcc, as well as via the package managers of major GNU/Linux distributions, including Debian, Ubuntu or Arch Linux.

References

[1] Ieee standard for ieee information technology - portable operating system interface (posix(tm)). *IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992)*, 2001.

[2] Joseph R Biden. Executive order on improving the nation's cybersecurity. 2021.

[3] Jonathan Brossard. Introduction to the witchcraft compiler collection. *Blackhat Conference, London, UK*, 2016.

[4] Jonathan Brossard. Introduction to procedural debugging through binary libification. In *18th USENIX Workshop on Offensive Technologies (WOOT 24)*. USENIX Association, August 2024.

[5] Jonathan Brossard. The witchcraft compiler collection. https://zenodo.org/doi/10.5281/zenodo.11405214, May 2024.

[6] Jonathan Brossard. The witchcraft compiler collection testsuite. https://zenodo.org/doi/10.5281/zenodo.11301409, May 2024.

[7] Mauro Capeletti. Unlinker: an approach to identify original compilation units in stripped binaries. 2016.

[8] Polona Car and Stefano De Luca. Eu cyber-resilience act — briefing eu legislation in progress — pe 739.259. *EPRS, European Parliament*, November 2023.

[9] Seth Carmody, Andrea Coravos, Ginny Fahs, Audra Hatch, Janine Medina, Beau Woods, and Joshua Corman. Building resilient medical technology supply chains with a software bill of materials. *npj Digital Medicine*, 2021.

[10] European Commission. Cyber resilience act - shaping europe's digital future. September 2022.

[11] Tis Committee et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2, 1995.

[12] Aleksandra Doniec. Converts a exe into dll. https://github.com/hasherezade/exe_to_dll, 2020.

[13] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020.

[14] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[15] Tommaso Frassetto. Raising the bar: Advancing mitigations against memory-corruption and side-channel attacks - doctoral dissertation, technische universität darmstadt. 2022.

[16] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. " not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404, 2011.

[17] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. A look at the design of lua. *Communications of the ACM*, 2018.

[18] Mahmood Jasim Khalsan and Michael Opoku Agyeman. An overview of prevention/mitigation against memory corruption attack. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, pages 1–6, 2018.

[19] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft*, 2013.

[20] Tavis Ormandy. Porting windows dynamic link libraries to linux. https://github.com/taviso/loadlibrary, 2017.

[21] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. Inference of robust reachability constraints. *Proceedings of the ACM on Programming Languages*, 2024.

[22] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. "weird machines" in elf: A spotlight on the underappreciated metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[23] Milan Stevanovic and Milan Stevanovic. The impact of reusing concept. *Advanced C and C++ Compiling*, 2014.

[24] Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2019.

[25] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 2005.

[26] Hui Xu, Yangfan Zhou, Jiang Ming, and Michael Lyu. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security. *Cybersecurity*, 2020.

[27] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 2023.

VIII. Appendixes

A. Witchcraft Script to assess CVE-2022-3602 in OpenSSL

```
print(" [*] Testing for CVE-2022-3602")

-- Input Arguments
teststring = "hello! -gr25faaaaaaaaaaaa"
decoded = calloc(80)
decodedlen = calloc(1,2)
memset(decodedlen, 0x14,1)

res = ossl_punycode_decode(teststring, 26, \
    decoded, decodedlen)

if res > 0 then
  free(res)
else
  print(" [*] Not vulnerable to CVE-2022-3602")
  exit(0)
end
```

B. Witchcraft Script to assess CVE-2022-3786 in OpenSSL

```
print(" [*] testing for CVE-2022-3786")

-- Input arguments:
teststring = "a.b.c.d.e.f.g.h.i.j.k.l.m.n.o.p.\
xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.\
xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.\
xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.\
xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--.xn--"

out = calloc(1,16)
outlen = calloc(1,2)
memset(outlen,0x10,1)

-- Trigger the stack overflow:
res = ossl_a2ulabel(teststring, \
    out, outlen)

if res == 0 then
  print(" [*] Not vulnerable to CVE-2022-3786")
  exit(0)
end
```