AUTOMATED SCANNING OF SOFTWARE PATCHES IN COMPILED BINARIES

Jamie Lim Jia Sin[1], Sng James[1], Khoo Wei Ming[2], Seah Rui Qi Daniel[2], Jonathan Cheng[2]
[1]NUS High School of Mathematics and Science, 20 Clementi Avenue 1, Singapore 129957
[2]DSO National Laboratories, 20 Science Park Drive, Singapore 118230

Abstract

Vulnerabilities in open-source software pose dangers to users because malicious actors can exploit these vulnerabilities to compromise software security. To mitigate these dangers, software patches are released to fix such vulnerabilities in compiled binary files.

We developed a tool to automatically scan for these vulnerabilities and their patches in stripped binary files. Our tool uses three methods: (1) static analysis of constants, (2) function emulation to analyse the output and control flow, and (3) analysis of global memory accesses.

We tested this tool on six well-known vulnerabilities across three different architectures. Static analysis successfully detected all the vulnerabilities and their patches, while the other methods had varying success. Generally, static analysis is the optimal method since it is fast, simple and applicable to many vulnerabilities and architectures.

Our tool can check if software contains previously disclosed vulnerabilities, preventing hackers from compromising these systems. Additionally, it can help analysts better understand stripped functions. This can be applied to other use cases such as discovering new vulnerabilities, improving the overall cybersecurity of open-source projects.

(172 words)

Introduction

When vulnerabilities are found in open-source software, they pose dangers to users since malicious actors can exploit these vulnerabilities to steal data, access computer systems or otherwise compromise software security. To mitigate such threats, software patches are released to fix or remove the vulnerable code. We aim to detect such known software patches in binary files.

One method to detect whether a piece of software has been patched is to reverse engineer the compiled binary file. This involves identifying the function containing vulnerable code, then checking if it has been patched. Analysing binary code is advantageous when the source code is unavailable or its version is unknown.

Hence, we wish to leverage machine code from the binary to identify functions and their vulnerabilities, using Ghidra (DeRosa, n.d.) to decompile and emulate the compiled binary. Ghidra is an open-source suite of software reverse engineering tools developed by the United States National Security Agency. Ghidra can decompile machine code into high-level code and supports many architectures. It also has robust scripting support and automatically extracts useful information from the binary, such as the different blocks of memory present.

Objectives

We aim to develop a robust tool to automatically detect whether a piece of software has been patched for a particular vulnerability, by identifying and checking the relevant functions. To accomplish this, we explore various function signatures that our tool may use: constants or strings in the code, outputs and control flow of the function during emulation, and memory accesses made by the function. Additionally, our tool should be general enough to support different architectures.

Related work

Other similar tools include Sibyl (Desclaux & Mougey, 2017), which uses the Miasm2 reverse engineering framework. It allows users to write several testcases, then emulates the binary to identify functions that match the expected output. A limitation of Sibyl is that it relies largely on Miasm2 for emulation. This means Sibyl only focuses on “pure” functions, whose behaviours are determined solely by their input and not external variables. However, many functions depend on global memory accesses or system calls, which Sibyl cannot identify.

Methodology

Our work can be split into three categories based on the signature used to identify the function and its patch. First, we explored static analysis of constants in the function. Next, we emulated the function with various testcases, comprising a set of inputs and expected output. This yields two other signatures: the function outputs or control flow, and the sequence of memory accesses.

Static Analysis

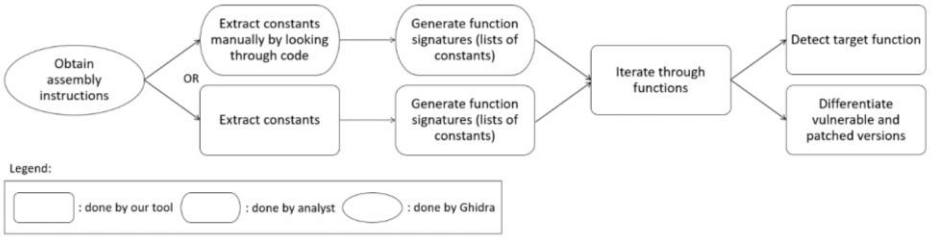

Figure 1: Flowchart of procedure for static analysis

For static analysis, we use Ghidra to obtain the assembly instructions from the binary. In each function, our tool iterates through these instructions to analyse which constants are present. These constants, which may be integers or strings, form the unique signature of the function, allowing us to detect the target function. The vulnerable and patched functions will often have different signatures, allowing us to differentiate them as well (see Figure 1). We consider constants as they are relatively easy to identify from the instructions.

A benefit of static analysis is that it is very fast, since we do not emulate the instructions, but merely extract constants from them. It is also architecture independent since the extraction is done from decompiled code. Due to the simplicity of this method, the time taken to manually analyse the function and obtain the signature is much shorter than for the methods below; alternatively, we have coded a script to automatically generate the function signature.

However, a downside to this method is that it is not very resistant to changes in the code; small patches that are implemented in the future may add or remove constants, which may affect the results. This also means attackers can easily work around this method by changing constants in the code while maintaining the code's functionality. We account for this by making our checking conditions as weak as possible, while still successfully identifying the function.

Emulation Outputs and Control Flow

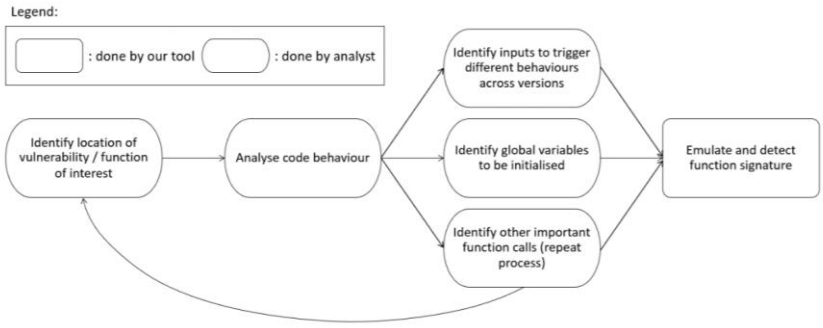

Figure 2: Flowchart of procedure for analysing emulation outputs and control flow

This method involves emulating the function with a specific set of inputs and pre-initialised global variables, then checking for signatures such as the function output or function calls made.

Firstly, we identify the vulnerable segment of the function of interest, which can be found in the released patch files or online articles. Next, we analyse the behaviour of the code in both the vulnerable and patched versions, taking note of the conditions needed for the vulnerable segment to be run. This allows us to identify the inputs or initialisation needed to trigger different behaviours between the two versions. This often requires us to identify other functions being called. Finally, our tool emulates the function and detects the signature we identified (see Figure 2).

Emulation is advantageous because it is more robust and resistant to changes in the code – if the purpose of the function does not change significantly, it likely requires a similar control flow (if-statements, function calls etc.). Use of emulation also makes this method architecture independent because the overall functionality of the code does not depend on the architecture used.

However, this method is time consuming because a deep understanding of the vulnerability, its patch and related functions is required. Moreover, some functions may be very difficult or even impossible to detect using emulation alone. They may be overly complicated with many nested function calls, or the vulnerable and patched versions may give highly similar signatures.

Memory Access Analysis

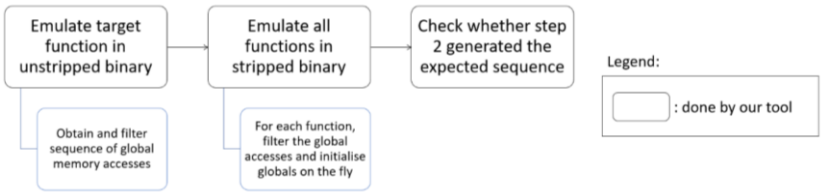

Figure 3: Flowchart of procedure for memory access analysis

Memory access analysis has three stages. First, we compile the source code of the patch of interest to obtain a known, unstripped binary. We then use our tool to emulate the target function in that binary to obtain the expected sequence of load and store instructions involving global memory, disregarding stack and heap accesses. This sequence is written to a text file, to be used in the next stage. Each element of the sequence consists of the type of access, the memory address being accessed and the value being loaded or stored (see Figure 3).

Our tool then filters this sequence to ensure only the crucial elements remain. In this process, the tool removes certain architecture-dependent loads and stores and ensure the sequence is the same across architectures. For example, MIPS may load a value containing the address of a variable it is trying to access. Since such extra memory accesses are not present in other architectures, our tool removes them. Similarly, ARM may load a value containing the address of another instruction. This is also not present in many other architectures and was thus removed.

Secondly, our tool emulates all the functions in the given unknown, stripped binary. Since we cannot obtain the addresses of global variables from the stripped binary directly, our tool initialises them on the fly during emulation. To do this, the tool scans through the expected sequence to find the first occurrence of each global variable. If this first occurrence is a load, we add it to a series *v* of "starting values" as shown in figure 4b. Next, in the stripped function, our tool checks whether each new memory access involves the global memory, using the same filter as above. Whenever the function loads from a new, uninitialised global variable, our tool initialises it with the next value from the series *v* (see Figure 4b).

Figure 4a:

Sequence from stripped binary (no initialisation)

Type	Address	Value
Load	X	0 (default)
Store	Y	2
Store	X	2 (wrong)
Load	Z	0 (default)
Load	X	2 (wrong)
Load	Y	2

Figure 4b:

Expected sequence from unstripped binary

Type	Address	Value
Load	A	1
Store	B	2
Store	A	3
Load	C	4
Load	A	3
Load	B	2

Series of "starting values", *v*

Type	Address	Value
Load	A	1
Load	C	4

Sequence from stripped binary (initialise on the fly)

Type	Address	Value
Load	X	1
Store	Y	2
Store	X	3
Load	Z	4
Load	X	3
Load	Y	2

Figure 4a and 4b: Initialising global variables from stripped binary on the fly. Figure 4a loads and stores incorrect values, showing the importance of initialising global variables on the fly.

If the unknown function is indeed the target function, this initialisation method will yield the same memory access sequence; otherwise, the sequence will likely be different. In our last step, the tool tries to match the elements of the unknown sequence to those of the expected sequence. We can compare the access type and value directly, but since memory addresses often differ across binaries, we only consider them "symbolically", disregarding the actual addresses. This means our tool maps each address in the expected sequence to a unique address in the unknown sequence (see Figure 5).

Expected Sequence

Type	Address	Value
Load	A	1
Load	B	2
Store	A	3
Store	B	4
Load	B	4

Unknown Sequence 1 (matched: A ↔ X, B ↔ Y)

Type	Address	Value
Load	X	1
Load	Y	2
Store	X	3
Store	Y	4
Load	Y	4

Unknown Sequence 2 (addresses do not biject)

Type	Address	Value
Load	X	1
Load	Y	2
Store	X	3
Store	Y	4
Load	Z	4

Unknown Sequence 3 (type does not match)

Type	Address	Value
Load	X	1
Load	Y	2
Store	X	3
Store	Y	4
Store	Y	4

Unknown Sequence 4 (value does not match)

Type	Address	Value
Load	X	1
Load	Y	5
Store	X	3
Store	Y	4
Load	Y	4

Unknown Sequence 5 (sequence length does not match)

Type	Address	Value
Load	Y	2
Store	X	3
Store	Y	4
Load	Y	4

Figure 5: Example of a known sequence with matched and unmatched unknown sequences (using symbolic addresses and filtered)

The main benefit of this method is that we need not obtain the addresses of global variables but can initialise them on the fly. This is better than control flow analysis, where we often had to search for and identify other functions to obtain the address of a global variable for initialisation.

However, this method is not as robust as control flow analysis as it depends on the number of memory accesses made. If the expected sequence is too short, this method usually cannot identify the function and will pick up false positives. This is hard to filter manually as stripped binaries do not contain function names and other symbols, so the analyst must look through the source code. New patches or versions can also change the sequence of memory accesses easily and in general, it is difficult to account for these small changes when comparing with the known sequence.

To account for some of these changes, we have implemented a small extension of this method, which can handle cases where (1) the order of loads in the unknown has changed, or (2) the unknown sequence contains an additional global variable. In these cases, our tool will try all permutations of the series of load values *v*. When matching the unknown and expected sequences, we only try to match within the sub-sequence for each variable, and not the relative order of memory accesses across variables.

Implementation of Our Tool

Our tool was written in Python and makes use of Ghidra's Jython API. We coded scripts for each method above, and we outline the main components they use below.

The Emulator class allows us to specify a function and a set of inputs, then run that function using the Ghidra emulator and returns the output. During emulation, we can either emulate all function calls, or skip them and specify the expected return values instead. While the emulator is running, it also allows us to analyse the code by setting breakpoints and stepping through the code instruction-wise. As we step through the code, we can view the contents of specific registers and the heap memory. This helps us analyse the control flow and behaviour of the function.

The PcodeHelper class is used in memory access analysis. P-code is a language designed for Ghidra as a common intermediate representation for different processor architecture instructions. Each assembly instruction in a binary is converted to a sequence of p-code operations. Ghidra emulates the p-code operations instead having to emulate the instruction sets of many different architectures.

During memory access analysis, our tool determines if each instruction is a load or store based on its sequence of p-code operations, by emulating (separately from Ghidra) the p-code at each instruction. When the tool detects a value being read from or written to the random-access memory, it adds the memory address and value involved to a list to be filtered later.

To initialise inputs and global variables, we make use of the following classes. The Testcase class contains the variables to be initialised and the expected output. The FunctionTest class stores information about the function, including its name, return type and Testcases. The StructFactory and Struct classes contain functions to recursively initialise custom structures, which are often found in the binaries we analyse. Lastly, the GlobalManager and HeapManager classes write to the global and heap memory respectively.

Our tool also allows us to specify our own application binary interfaces (ABIs) to support various architectures. Each ABI contains information about the architecture, including important registers like the input, output and stack registers. It also contains information about how inputs, global variables and return values should be handled. In our testing, we specified x86, MIPS and ARM ABIs to initialise the necessary pointers or register values before emulation.

Results

We tested our methods on some well-known vulnerabilities, as shown in the table below. We successfully tested all three methods on Bash Shellshock for x86-64, x86-32, MIPS and ARM. For the other vulnerabilities, the methods indicated below work for x86-64.

Common Vulnerabilities and Exposures (CVE) Identifier	Common Name	Static Analysis	Emulation Outputs / Control Flow	Memory Access Analysis
CVE-2014-0160	OpenSSL Heartbleed	✓	✓	✗
CVE-2014-6271	Bash Shellshock	✓	✓	✓
CVE-2016-5195	Linux Kernel Dirty COW	✓	?	*
CVE-2022-0847	Linux Kernel Dirty Pipe	✓	?	✓
CVE-2017-3167	Apache Authentication Bypass	✓	?	✗
CVE-2019-0215	Apache Access Control Bypass	✓	?	*

Table 1: Vulnerabilities analysed and methods used to identify them (✓: can identify and differentiate vulnerable and patched functions; *: can identify function only; ✗: probably impossible to identify function; ?: did not test)

The software versions we tested the above methods on are as follows:

OpenSSL Heartbleed: 0.9.8, 1.0.0, 1.0.1, 1.0.2
Bash Shellshock: 3.0 – 3.2, 4.0 – 4.3
Linux Kernel Dirty COW: 4.8
Linux Kernel Dirty Pipe: 5.6, 5.10, 5.15
Apache Authentication Bypass: 2.2, 2.4
Apache Access Control Bypass: 2.4

Static Analysis

The following is an example of a static analysis procedure we have developed for the Apache Authentication Bypass vulnerability. The procedures for other vulnerabilities are similar.

We first try to identify the function of interest, `ap_get_basic_auth_pw`, by checking if any function contains the error message string `"client used wrong authentication scheme: %s"` but not `"user %s: authentication failure for "%s": Password Mismatch"`. If there is no such function, we conclude the binary is not vulnerable. Otherwise, within this candidate function, we try to detect the string `"AP_GET_BASIC_AUTH_PW_NOTE"`. This string was introduced in the patch for this vulnerability, so if it exists, then the function is patched.

This method may declare that a function is a vulnerable version of `ap_get_basic_auth_pw`, although that function is not even present in the binary. However, this is simple for the analyst to verify. Our method will not make the dangerous claim that a vulnerable version is patched.

Emulation Outputs and Control Flow

The following is an example of a procedure we developed to detect OpenSSL Heartbleed via emulation. The procedure for Bash Shellshock is more complicated and not detailed here.

Firstly, we identify the function of interest, `dtls1_process_heartbeat`. It takes in an input of a custom data type, `SSL *s`, then makes a call to `s -> msg_callback`. Since our tool can initialise input values, we use it to set `s -> msg_callback` to `1` and place a breakpoint on the address `1`. If the emulator reaches the breakpoint, we can be fairly certain that we have found the function of interest because the address `1` will not normally be a valid instruction address.

Next, we detect if `dtls1_process_heartbeat` is patched. The vulnerable function always runs the external function `CRYPTO_malloc`, but the patched function checks certain conditions before running `CRYPTO_malloc`. We can set testcases for our tool to identify standard C library functions such as `malloc`, which is similar to `CRYPTO_malloc`. Hence, our tool can identify and get the address of `CRYPTO_malloc`. We then set the input `s` such that the conditions in the patched version will fail, allowing us to differentiate the two versions based on whether `CRYPTO_malloc` is called.

We did not test this method for the remaining 4 vulnerabilities as the emulation method is time-consuming, especially for vulnerabilities that are highly similar to their patched versions.

Memory Access Analysis

For Bash Shellshock and Dirty Pipe, we can identify the function via the memory access sequence because it has many global accesses, so its sequence is different from other functions in the binary. We can also differentiate the vulnerable and patched functions via the memory access sequence, by initialising the inputs to force different control flows.

For OpenSSL Heartbleed and Apache Authentication Bypass, it is unlikely we will be able to identify the function via memory access analysis because few global variables are used, so the memory access sequences are very short. Thus, this method detects many false positives.

For Dirty COW, our tool has managed to identify the function of interest via this method. Analysing the decompiled code, we believe it is theoretically possible to identify the variables that need to be initialised in order to differentiate the vulnerable and patched control flows. However, this would require significant time and effort on the analyst's part and hence is not recommended.

For Apache Access Control Bypass, the patch applied is an additional external function call (`SSL_set_verify`), for which our tool is unable to obtain the memory access sequence. Hence, the sequences will be the same before and after the patch, making it impossible to differentiate them.

Applications

This tool makes it easier for people with little background in vulnerability analysis to detect software vulnerabilities by simply running our scripts. This prevents hackers from compromising

their systems by exploiting such vulnerabilities. Our tool can also help check if new software releases contain any previously disclosed vulnerabilities, because it is unlikely to give false negatives and flag a vulnerable binary as safe.

Commented [SJ1]: Changed old to this to follow abstract, also do we need to add the "improve cybersecurity of open-source software" part

Moreover, our tool can initialise memory prior to emulation, helping analysts understand stripped functions more deeply with dynamic analysis. This feature can be applied beyond function or vulnerability identification, to other use cases involving memory accesses, such as discovering new vulnerabilities. This can improve the overall cybersecurity of various open-source projects.

Future Work

With more time, we would test our methods on more architectures, vulnerabilities and software versions. We would also like to work on a more robust method of matching the known and unknown memory access sequences, which would be able to account for minor changes due to patches. For example, we could try to identify the extraneous or removed memory accesses in the unknown sequences, based on the order in which the addresses appear in the sequences.

Another way we can overcome the limitations of the methods discussed above may be to combine different elements of each method. For example, we might be able to perform static analysis to identify the function, then process and analyse the sequence of memory accesses to compare and differentiate the vulnerable and patched functions.

We also hope to improve the user experience when using our tools by creating a frontend user interface. This would allow users to use our tool without having to modify our configuration code or use the Ghidra interface, which can be quite overwhelming for first-time users.

Conclusion

In this study, we have developed a tool to automatically scan for software vulnerabilities and their patches in stripped binary files. Our tool uses three methods: static analysis, analysis of emulation outputs and control flow, as well as memory access analysis. We have tested this tool on six well-known vulnerabilities across different architectures and can successfully determine whether each of these vulnerabilities exists or has been patched in their respective binaries.

For most purposes, static analysis is the optimal method of scanning for vulnerabilities since it is fast, simple and applicable to many vulnerabilities and architectures. However, static analysis should ideally be used on the original binaries; if malicious actors have modified the constants, there is a risk of inaccurate results, so the other methods might perform better.

Acknowledgements

We would like to thank our supervisors, Dr Khoo Wei Ming, Mr Seah Rui Qi Daniel and Mr Jonathan Cheng, for mentoring us throughout this project. We would also like to acknowledge the support of our teachers, in particular Mrs Phylliscia Lee, Dr Chiam Sher-Yi and Dr Maury Julien Jean Pierre. Lastly, we would like to thank our family members for their continuous support.

References

DeRosa, A. (n.d.). *Machine Emulation with Ghidra*. Retrieved from Syscall7: https://syscall7.com/machine-emulation-with-ghidra/

Desclaux, F., & Mougey, C. (17 June, 2017). *Miasm2 Reverse engineering framework.* Retrieved from REcon: https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-miasm.pdf