Can Architecture Design Help Eliminate Some Common Vulnerabilities?

Strahinja Trecakov*, Casey Tran†, Abdel-Hameed Badawy‡, Nafiul Siddique‡, Jaime Acosta§, Satyajayant Misra*
*Computer Science, New Mexico State University
†Computer Science, Humboldt State University
‡Klipsch School of Electrical and Computer Engineering, New Mexico State University
§Army RDECOM ARL, White Sands Missile Range

***Abstract*—As technology improves in size and the number of smart devices increases, security in personal devices undoubtedly becomes an important aspect of todays life. However, the complexity in hardware and software systems expose vulnerabilities in security. Vulnerabilities may exist in many layers of systems and would require a specific inputs or events to trigger it. Discovery of vulnerabilities require significant time and also system specific knowledge, and even then some are difficult to patch.**

In this paper, we study open source tools for finding potential vulnerabilities and represent the advantages and disadvantages in their use. We present *HardVul*, a vulnerability checking tool which can be run on any architecture and reports which vulnerabilities were found from our testbed.

I. Introduction

Today, many software producers and users want to have bug-free products. However, due to the complexity of software and expanding nature of software industry, this is basically impossible. Additionally, software bugs cannot be resolved completely with software updates–as updates resolve previous versions' bugs they introduce new bugs that makes the software vulnerable.

Vulnerabilities can be present and triggered by any layer of a computer system (Figure 1). However, each vulnerability has a certain risk level depending on what can be compromised [1]. There are several efforts/tools for finding vulnerabilities, but they are new and not well documented. As example, the Department of Homeland Security and the National Cyber Security Center recently made efforts to document common security weaknesses that lead to software vulnerabilities. They created the Common Weaknesses Enumerator (CWE) list with all the weaknesses found so far [2].

In this paper, we present our benchmark suite that is used to evaluate the performance of different architectures. This was primary developed to test our Operating System Friendly Microprocessor Architecture (OSFA), however, we made it applicable to other architectures for comparison studies of *e.g.*, reduction of impact, overhead, and performance. The benchmark contains binaries with traditional memory corruption vulnerabilities in *e.g.*, MITRE CWE list [2]. In addition to the vulnerable binaries, the benchmark contains code that exploits the vulnerabilities.

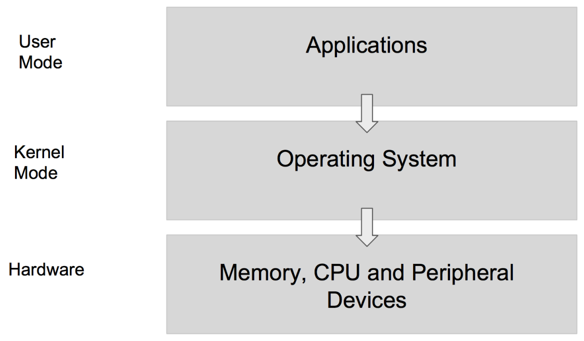

Fig. 1: Typical representation of layers in computer system

The benchmark suite was completed in three phases. First, we identified vulnerabilities from CWE list. Then we developed exploits for vulnerability binaries, and finally we evaluated our results.

II. Background

In this section, we briefly describe OSFA, computer system layers, their cause of vulnerability and necessary security.

A. OSFA

The OSFA is a newly patented architecture that provides faster context switching because of it pipeline configuration as well as its cache banks [3].They can be swapped between execution and memory pipeline which make this fast context switching possible. Each cache bank and memory address contains Unix file permission bits that provide better computer security.

B. Computer System Layers

In Figure 1, we show three basic layers. The top layer consist of the applications with which the user interacts. The Operating Systems (Kernel) level is the computer's operating system core that controls and interacts with the central processing unit (CPU), memory, and hardware. Below the OS layer is the hardware system with almost all devices [4].

C. Cybersecurity and Computer Security

Cyber-security is an area of national security and computer systems security is a universal requirement in all systems at all times. The prevalence of the recent attacks that target political entities and figures, financial institutions, and the Internet service providers makes research and development of computer architectures that are designed to fend off attacks a priority [5].

Computer security is a universal problem across computer domains. What is needed is better computer security with less microprocessor, OS, and application software overheads. The patented OSFA creates secure sandboxes for each executing process with very little overhead [3].

D. Vulnerabilities

Vulnerabilities can cause major damage to a broad spectrum of entities from the government, corporations, and to the common man [6]. Unpatched software vulnerabilities are dangerous since they may allow hackers to get into the system through a backdoor and steal sensitive information [7]. There are many types of software vulnerabilities and they are found in code, design, or system architecture. Some of them are buffer overflows, authentication issues, error handling, string formating,interaction error, data handling, and many more.

III. Related Work

A. CWE

The CWE list is a formally documented list of all known software security weaknesses in code, design and architecture. It is also used to help with vulnerability identification, migration and prevention. This list was developed by MITRE's Common Vulnerabilities and Exposures(CVE) team, a not-for-profit organization that operates research and development centers sponsored by the federal government working for the public interest and no commercial interest [2]. Their organization structures each software vulnerability in CWE such that each may be referenced according to its own vulnerability type. Comprised of 722 and 769 CWEs for research and development concepts, respectively, these vulnerabilities can be classified under a hierarchical structure such that we may use the CWEs at the generalized and higher levels of the structure to provide a broad representation from a ”parent” weakness to test on [8], [9].

B. Angr

In this section we present Angr, a binary analysis tool built under a python framework. Made up of several sub projects, it is comprised of an executable and library loader (CLE), a library describing various architectures(archinfo), a python wrapper around the binary code lifer VEX(PyVEX), a code simulation engine(SimuVEX), and a data backend to abstract away differences between static and symbolic domains(Claripy) [10], [11]. These state-of-the-art binary analysis tools serve to make Angr one of the first open-source binary analysis frameworks for building upon already existing implementations of binary analysis techniques. Resulting from these efforts is a software testing platform devised with an intention towards systematizing the field of binary analysis. As a result, different tools may be openly used and compared with each other in order to further research efforts in cyber-security.

Angr was a good candidate for testing our architecture for vulnerability exploits because it is capable of being a symbolically assisted fuzzing tool– ie, that it is suited to be able to find deeper exploits. Moreover, it is open source and meant to be used and built upon by the software security community. Its purpose is explicitly to provide a tool that will be useful in reproducing, improving, and creating binary analysis techniques. Adding to the security community, this in turn means that our application of Angr may also be able to test other architecture simulations combining the static and dynamic techniques implemented in Angr.

Individually, Static and dynamic analysis techniques alone each have their respective tradeoffs. Static analysis alone is good for finding *concretized* general inputs but cannot provide to specific and traceable paths. However, dynamic analysis techniques are good for finding specific inputs and tracing. These techniques when put together are a called ”concolic” analysis due to the concrete aspect of a static technique and the symbolic aspect of a dynamic execution. As we shall see in the next section, Angr acts as the symbolic tracer of Driller, being able to guide its fuzzing engine into deeper paths and hence finding deeper vulnerabilities. But before moving on to the next section, it is worth mentioning the type of Fuzzer that Driller utilizes as the primary tool for producing general inputs for very high automated code coverage.

C. Driller is AFL + Angr

In the previous section we discussed the advantages of Angr as a concolic execution tool utilizing symbolic and concrete techniques of binary analysis. In this section we will present an overview of Driller. Driller is an excavation tool that makes use of the American Fuzzy Lop (AFL) as its fuzzing engine and Angr as its concolic style execution tool. Fuzzing is an automated software testing technique that excels at producing general inputs. In regards to AFL, the fuzzing program automatically creates and mutates inputs along a path until it finds a vulnerability [12].

Working in tandem with Angr as its symbolic execution engine, American Fuzzy Lop (AFL) is utilized in Driller as its fuzzing engine . This results for a powerful binary analysis tool that is capable of ”Symbolic-assisted fuzzing” [10]. Where AFL is used to quickly find solutions for general inputs and code coverage from its genetic mutation algorithm for finding paths which are ”interesting”, Angr is good at finding solutions for specific inputs that a program application may require in order to find vulnerabilities that may exist deeper within the code path. As a result, deeper vulnerabilities are able to be discovered that would otherwise not be possible by using one of the techniques alone. Secondly, path explosion from using a dynamic symbolic analysis is mitigated due to the Fuzzer thus resulting in more scalability. Furthermore, the incompletenesses of fuzzing from what it declares as an ”uninteresting path” are more thoroughly executed through Angr as the concolic execution tool [13], [14]. Now we shall see how Driller works.

Angr and AFL both utilize static analysis techniques. The difference is that Angr also uses dynamic symbolic executions in conjunction with a symbolic tracer to retrieve paths. So, while Angr has a static analysis state running parallel with its dynamic symbolic executions to generate concretized inputs from the set of constraints resulting from the dynamic executions, AFL generates concretized inputs from its genetic mutation algorithm [10], [13]. The former results in more specific and ”harder” to find pathways while the Fuzzer offers fast functional code coverage . Together, Driller makes deeper excavation possible.

Driller works like this, first it runs AFL and generates static inputs until it cannot find anymore ”valuable/interesting” paths or stops at a timeout specified either by default or the user. For each interesting input generated by the genetic algorithm of AFL, Angr first symbolically executes the input and gets the trace of basic blocks taken by the input from constraints used to generate them by the static analysis state running parallel. Path explosion is thus mitigated as a result of AFLs algorithm and interesting and uninteresting inputs are excavated more deeply with the concolic analysis of Angr.

IV. Design Model and Evaluation

```
char pass[] = "abcd";
int validate_user() {
  char buff[5];
  printf("Enter your password:\n -> ");
  gets(buff);
  return !strcmp(buff, pass);
}

int main(int argc, char *argv[]){
  if(validate_user()){
    printf("Your password was correct\n");
  }
  else{
    printf("Your password was not correct\n");
  }
  return 0;
}
```

Fig. 2: CWE-120 Buffer Overflow based example [8]

A. Design Model

In this section, we present the first two phases of our benchmark. We used the CWE list to identify common vulnerabilities that will help us test different architectures. We picked a couple of known vulnerabilities from each category such as stack-based buffer overflow, heap-based buffer overflow, integer overflow, incorrect pointer scaling, and expired pointer dereferences. In this paper, we present a vulnerability based on CWE-120: Buffer Copy without Checking Size of Input. This is a Classic Buffer Overflow example where a program allows an input larger that the buffer size (Figure 2).

During the second phase, we studied two open source tools: Angr and Driller. Driller is a great candidate because it uses concolic execution to find all paths in a program [13]. Unfortunately, we were not able to modify Driller and make it work on the broader range of platforms. Some of the reasons are that Driller has a poor support for Linux/ELF binaries since it was primary built for DARPA’s Cyber Grand Challenge and supports only DECREE binaries [15]. Secondly, the Angr’s SimProcedures module does not support all of C libraries and we can not rely on the output. Moreover, Driller only detects segmentation faults and it will require a lot of manual analysis and modifications in order to recognize other vulnerabilities [16]. After releasing all this, we decided to develop simple vulnerability analysis tool.

First, we developed exploits for vulnerability binaries by writing source code for all vulnerabilities that are in our test-suite. We also gathered all the object dumps (objdumps) and traces of our test cases to be able to say when we have a memory corruption and when not. Then we wrote a simple program that runs our test-suite with inputs that trigger vulnerabilities, get information dumps and compares them with ones that do not trigger vulnerabilities.

```
0x7fffffffd9b0:0x00 0x00 0x00 0x00 0x00 0x00
    0x00 0x00
0x7fffffffd9b8:0x00 0x00 0x00 0x00 0x00 0x00
    0x00 0x00
0x7fffffffd9c0:0xe0 0xd9 0xff 0xff 0xff 0x7f
    0x00 0x00
0x7fffffffd9c8:0xb7 0x06 0x40 0x00 0x00 0x00
    0x00 0x00
```

Fig. 3: Representation of the stack pointer before inputed password

B. Evaluation

After gathering all the outputs, we analyzed results in few ways. First, we made sure that our HardVul tool works properly, by comparing traces of our tests with inputs that do and do not trigger vulnerabilities. If we look deeper into (Figure 3), we can see 32 bytes representation of stack pointer before any password was provided. However, (Figure 4) represent stack pointer after ”AAAAAA” was inputed as a password. Clearly, even if the character array buff was declared with size 5, input bigger than 5 is being accepted and stored. This is buffer overflow, where the part of the input that does not fit in the buffer keeps being written in adjacent memory addresses. This

```
0x7fffffffd9b0:0x41 0x41 0x41 0x41 0x41 0x41
    0x00 0x00
0x7fffffffd9b8:0x00 0x00 0x00 0x00 0x00 0x00
    0x00 0x00
0x7fffffffd9c0:0xe0 0xd9 0xff 0xff 0xff 0x7f
    0x00 0x00
0x7fffffffd9c8:0xb7 0x06 0x40 0x00 0x00 0x00
    0x00 0x00
```

Fig. 4: Representation of the stack pointer after inputed "AAAAAA" an a password

event results in program malfunction or segmentation faults. If the input is carefully selected, targeted addresses value can be altered to jump to places in the program that normal execution would not execute.

Then we preformed our benchmark testing on few different architectures using gem5 simulator. This simulator supports variety different Instruction Set Architectures(ISA), such as ARM, Alpha, SPARC, and x86. It also has two different modes that are full system and system call emulation. Difference between these two modes is that the full system will boot Linux and allow system calls that incorporate OS into simulation. On the other hand, system call emulator does not allow that [17].

Our current test-suite contains over 100 vulnerabilities that are grouped based on the CWE list. The groups are stack based buffer overflow, heap based buffer overflow, buffer underwrite, buffer overread, buffer underread, integer overflow, and integer underflow. Moreover, some of vulnerabilities in our test-suite may also trigger other weaknesses that are documented in the CWE list.

After performing our benchmark testings using gem5's full system mode on ARM, Alpha and x86, we noticed that program memory traces are slightly different because of different architectures. However, they all had the same results when it comes to vulnerability test.

V. Conclusions and Future Work

After performing both manual and automatic analysis on a couple of different architectures using our HardVul tool, we found some interesting data that gives us more confidence that architecture design can help prevent some vulnerabilities. However, this may open space for some other/new bugs.

Once the OSFA simulator is done, we will expand this research by testing our HardVul tool on it and include more vulnerabilities that we think OSFA design can prevent from. We will try to make our tool user friendly, and make it an open source tool.

On the other hand, the research done in this paper needs to be expanded to look more deeply into this problem and try to include as much vulnerabilities as possible. Also it would be nice to create a test-suite that includes each known vulnerability category.

References

[1] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 771–781.

[2] R. A. Martin and S. Barnum, "Common weakness enumeration (cwe) status update," *Ada Lett.*, vol. XXVIII, no. 1, pp. 88–91, Apr. 2008. [Online]. Available: http://doi.acm.org/10.1145/1387830.1387835

[3] P. Jungwirth and P. La Fratta, "Os friendly microprocessor architecture: Hardware level computer security," in *SPIE Defense+ Security*. International Society for Optics and Photonics, 2016, pp. 982 602–982 602.

[4] Y. Shi, D. V. Murillo, S. Wang, J. Cao, and M. Zheng, "A command-level study of linux kernel bugs," in *Computing, Networking and Communications (ICNC), 2017 International Conference on*. IEEE, 2017, pp. 798–802.

[5] P. Jungwirth and A.-H. Badawy, "Cybersecurity and the osfa architecture."

[6] O. H. Alhazmi and Y. K. Malaiya, "Application of vulnerability discovery models to major operating systems," *IEEE Transactions on Reliability*, vol. 57, no. 1, pp. 14–22, 2008.

[7] A. M. Algarni and Y. K. Malaiya, "Most successful vulnerability discoverers: Motivation and methods," in *Proceedings of the International Conference on Security and Management (SAM)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 1.

[8] "Cwe list," https://cwe.mitre.org/data/published/.

[9] "Nvd list," https://nvd.nist.gov/vuln/categories/.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[11] "Angr documentation," http://angr.io/.

[12] "Fuzzer," https://github.com/shellphish/fuzzer/.

[13] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[15] N. Lee, "Darpas cyber grand challenge (2014–2016)," in *Counterterrorism and Cybersecurity*. Springer, 2015, pp. 429–456.

[16] "Driller source code," https://github.com/shellphish/driller/.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.