CBTracer: Continuously Building Datasets for Binary Vulnerability and Exploit Research

Yukun Liu
Tsinghua University
liuyukun16@gmail.com

Zhuge Jianwei
Tsinghua University
zhugejw@cernet.edu.cn

Chao Zhang
Tsinghua University
chaoz@tsinghua.edu.cn

ABSTRACT

Vulnerability discovery and exploiting are critical to software security. Emerging *intelligent* vulnerability discovery solutions usually require a large number of training data. Studying exploits also requires a set of existing exploit samples. As a result, building a dataset for vulnerability and exploit research is necessary. In this paper, we present CBTracer, able to catch real-time I/O traffic of target applications and monitor their runtime executions, to build an evolving dataset for kinds of security analysis, including vulnerability discovery and exploit generation. CBTracer is a lightweight framework designed to be deployed easily in various CTF competitions by different organizers to build a bigger dataset. We used CBTracer to collect data from CGC challenges and deployed it in several real-world CTF challenges, showing that it could efficiently collects security-related data.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software security engineering**;

KEYWORDS

dataset, vulnerability, exploit

ACM Reference format:
Yukun Liu, Zhuge Jianwei, and Chao Zhang. 2018. CBTracer: Continuously Building Datasets for Binary Vulnerability and Exploit Research. In *Proceedings of The 1st Radical and Experiential Security Workshop, Incheon, Republic of Korea, June 4, 2018 (RESEC'18),* 7 pages.
https://doi.org/10.1145/3203422.3203429

1 INTRODUCTION

Vulnerabilities play an important role in system security. Despite the numerous efforts on improving the security of software, vulnerabilities still pervade in software, endangering the cyberspace. As a result, vulnerability discovery is still an active and important research field. Moreover, exploiting vulnerabilities becomes more and more sophisticated, making it challenging for defenders to catch up and react accordingly.

Aside from manual analysis, automated vulnerability discovery solutions generally fall into four categories: static analysis, dynamic analysis, smart fuzzing and symbolic execution. Static analysis solutions such as IST4 [53] and Clang Static Analyzer [25] could find many vulnerabilities with a low overhead, but a high false positive ratio. Dynamic analysis solutions, e.g., AddressSanitizer [44], TaintCheck [32] and MemCheck [45], could provide more accurate results, but require inputs to trigger vulnerabilities (or at least exercise vulnerable paths). Symbolic execution, e.g., KLEE [4] and S2E [6], is a popular solution in both vulnerability discovery and exploit generation. However, it faces two challenges, i.e., path explosion and complex constraint solving, limiting its scalability. Smart fuzzing, e.g., AFL [57] and Peach [15], is the most popular solution nowadays, widely studied in both industry and academic communities. Code coverage is a key factor to fuzzers. Solutions like SAGE [18], Driller [50] and VUzzer [42] improve fuzzers with other analyses, to get better code coverage and trigger more vulnerabilities.

Due to these limitations, researchers are still looking for new vulnerability discovery solutions. Inspired by the huge success of artificial intelligence in many fields, security researchers also seek the help of machine learning, deep learning and big data to discover vulnerabilities, e.g., [19, 20, 38, 56]. However, these solutions usually require a large amount of training data, which is hard to get, since existing vulnerability information are not well organized or not open to the public.

On the other hand, defenders are trying to catch up with attackers on vulnerability exploiting, in order to react to new threats. However, exploiting vulnerabilities becomes more and more challenging due to the increasing defenses. Researchers have tried automatic exploit generation, e.g., AEG [2] and Mayhem [5]. DARPA even launched a two-year program Cyber Grand Challenge (CGC) [11], calling for automated solutions. The results showed that state-of-art exploit generation solutions still demand a lot of engineering effort. In order to better understand exploits, to develop either advanced exploits or defenses, we have to learn from existing exploits. However, it requires a large number of exploit samples, which is hard to get too, since exploit samples are valuable.

As a result, building a dataset to collect security-related features is necessary and helpful, for vulnerability discovery, exploit generation and defense development.

If we could capture traffic related to attacks, we could build a comprehensive dataset of vulnerability exploits. Based on this observation, we propose a framework to capture exploit traffic as well as benign traffic, together with runtime monitoring on applications. With these collected metadata, we could continuously build a dataset with rich security information, able to support further analysis.

RESEC'18, June 4, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5757-9/18/06…$15.00
https://doi.org/10.1145/3203422.3203429

There are some existing solutions that are capable of capturing attack traffic, including honeypots [39, 40] and sandboxes [16]. They could simulate client or server side applications, usually with virtualization techniques, and capture malicious traffic or files. However, these solutions are usually very heavy weight, and could be easily detected by attackers [22, 23, 59]. There are hundreds of CTF competitions held every year. It will be a large security-related data if we could collect the data from these CTF challenges. In order to build a dataset for vulnerability and exploit analysis, especially for CTF organizers, we need a light-weight framework to capture network traffic and monitor runtime execution status. Moreover, the instrumented system should not affect vulnerability exploits' execution and should not be perceived directly.

In this paper, we present CBTracer, able to catch real-time I/O traffic of target applications and monitor their runtime executions, to build an evolving dataset for kinds of security analysis, including vulnerability discovery and exploit generation. We implemented CBTracer to continuously build datasets for security analysis from CGC and CTF challenges. CBTracer is a light weight framework which can capture traffic and extra runtime information of target program without modifying the program. Compared with other solutions, it is easy to be deployed with CTF challenges. CTF Organizers only need to make little change in xinetd configuration. The experience shows that CBTracer can collect security-related data effectively with almost no effect on CTF players.

The remaining part of this paper is organized as follows: Section 2 discusses related work. Section 4 and 5 present the architecture and implementation of CBTracer. The experimentation and evaluation results are discussed in Section 6. Section 7 presents the dataset. It concludes with challenges and future work in Section 8.

2 RELATED WORK

2.1 Exploit Logging

There are several existing solutions capable of capturing attack traffic or detect malicious activities. Honeypot is a solution. As explained in [40], a honeypot is a closely monitored network decoy which allows in-depth examination of adversaries during and after exploitation. It could capture the traffic and detect malicious content from the traffic.

Honeyd [39, 40] is the most popular framework for virtual honeypots. It simulates virtual computer systems at the network level by simulating the networking stack of different operating systems and providing arbitrary routing topologies and network services. It could detect malicious behavior with the help of third-party solutions, e.g., ReVirt [14] and network intrusion detection systems (IDS) [24, 37]. Correspondingly, PhoneyC [30] is the client-side honeypot, able to simulate victim clients by performing dynamic analysis of scripts in web pages to analyze malicious web sites.

Connections to honeypots are usually marked as malicious, since benign users would not access these services. Additional tools are required to analyze the attacks captured by honeypots in detail. ReVirt [14] is one of such tools. It logs all non-deterministic events that can affect the execution of the virtual-machine process, and replays the long-term execution instruction-by-instruction. In addition to its heavy weight nature, a honeypot could not capture benign traffic and limit its ability to build a comprehensive dataset.

Sandboxing is another solution to capture exploits. There are several products good at capturing real world exploits, for example, Fireeye's malware analysis sandobx [16]. Similar to honeypots, sandboxes are usually built based on virtualization, and are heavy weight. Moreover, they could be detected by attackers [22, 23, 59], and thus fail to capture some advanced exploits.

PANDA [54] is a whole-system dynamic analysis engine based on QEMU. It can be used to record and replay executions for malware analysis. It is also based on virtualization, and developers need to do extra work to filter the traffic and executions.

2.2 Existing Datasets

There are also some existing datasets related to vulnerabilities and exploits.

National Vulnerability Database (NVD) [35] is a comprehensive cyber security vulnerability database that integrates publicly available U.S. government vulnerability resources and provides references to industry resources. Bugzilla and MFSA are two other widely used vulnerability disclosure sources, as used in [46]. There are some other representative vulnerability sources, e.g., [7, 31, 47], as used in [34]. However, these vulnerability datasets usually lack of detail vulnerability information, e.g., proof-of-concept samples, detail vulnerable function and instruction information. NIST Juliet test suite [43] is a collection of crafted test cases, demonstrating different types of vulnerabilities. However, the number of samples is limited.

RIPE (Runtime Intrusion Prevention Evaluator [55]) is an exploit dataset consisting of 850 buffer-overflow attacks against popular defense mechanisms. The main purpose of RIPE is to provide testbed for developers of defense mechanisms. Exploit-db [36] and Metasploit [41] also provide some known exploits against known vulnerabilities. However, these exploit datasets are usually small and could not cover many types of vulnerabilities.

Kyoto2006+ [49] is another dataset created by several honeypots, consisting of 3-year real traffic data (Nov. 2006 - Aug. 2009). This dataset is valuable for network IDS evaluation.

2.3 Interpret Dataset

Some emerging work utilize known vulnerabilities information to predict existence of new vulnerabilities, using machine learning algorithms. For example, VDiscover [20] extracts function calls and their arguments as features, and tries several machine learning algorithms to predict whether new test cases could trigger a vulnerability or not. Since the dataset used in this paper is small, and the extracted features are not intrinsic to vulnerabilities, the precision of this solution is not good though.

There are other works on exploit classification, which aims at identifying the families of exploits and profile attackers when possible. Nguyen et al. have introduced various machine learning techniques for IP traffic classification briefly. [33], Kuldeep Singh et al. tried five machine learning algorithms, and figured out the most effective one for IP traffic classification. [48]

Other exploit classification solutions include vulnerability-specific and network-level analysis.

Some vulnerability-specific analysis solutions classify exploits by focusing on the workflow of how vulnerabilities are triggered. [8,

21, 26, 58] However, they are too slow to analyze real-time attacks. SeismoMeter [13] presented a novel solution which generates a succinct data representation to characterize the captured exploit.

Network-level analysis characterizes attacks at the network level. WOMBAT[10] classifies attack events using statistical characteristics such as source IP, attack time and armies of zombies. However, They are not as accurate as vulnerability-specific analysis because these characteristics usually are not intrinsic properties of exploits. Attackers can bypass detection through evasion or obfuscation[3, 17, 52].

3 DATASET

We would like to build a dataset to support both vulnerability discovery, exploit generation and other security research. It will cover a rich set of information related to security properties.

In this section, we will discuss the differences of different data sources and the design principle of the data in the dataset.

3.1 Data Sources

To build a dataset for vulnerability and exploit research, we need to interact with target applications using both normal and malicious request or monitor the interaction with others. During the interaction, we should catch real-time I/O traffic of target applications and record their runtime executions. For the target applications, there are three types we can choose from.

3.1.1 Real-world software. The traffic of the real world is realistic, with both known and unknown exploits. The real-world software is so complicated that there are various ways of interaction. We have to monitor different way of interaction to record the traffic. The benign traffic is much more than malicious traffic in the real world, and it is difficult to get a ground truth and label malicious traffic in the real world.

3.1.2 CTF Challenges. Exploit challenges in CTF game abstract the vulnerabilities in realistic software. To focus on exploit technique, the way of interaction is usually simplified by interacting with standard input/output. And most of exploit challenges are ELF binaries running in Linux. There are two popular format in CTF, jeopardy and attack-defense. challenges in jeopardy game are deployed in server, waiting for players to exploit them and get flags. In attack-defense game, each team maintains a challenge instance which is the same in the beginning. During attack-defense game, organizers need to launch availability checks to ensure the challenges of each team work normally.

Nowadays, hundreds of CTF competitions are held every year. There will be a large amount of data if the organizer records the traffic and runtime executions of challenges. Or we can build dataset with the released challenges and exploits after the game. However, there are some flaws of collecting data from CTF challenges. Most of traffic in CTF game is malicious especially in jeopardy mode. In attack-defense game, availability checks can be considered as benign traffic, but they are still quite less than malicious traffic.

3.1.3 CGC Binaries. DARPA Cyber Grand Challenge(CGC) binaries are custom-made programs that contain various vulnerabilities. Combined with CGC Qualifying Event(CQE) and CGC Final Event(CFE), there are around 200 challenges. For each binary, there are some polls and proof of vulnerability(PoV). Polls are similar to availability checks or SLA checks in attack-defense games, CGC polls interact with a challenge set to demonstrate and enumerate behavior that must not be patched out. PoV is an exploit. It can either negotiate and demonstrate register control (a Type 1 PoV) or disclose magic page data (a Type 2 PoV).

It is natural to get ground truth using CGC binaries as target programs, while it is difficult to get ground truth from realistic traffic. The polls can be considered benign interaction and the PoVs can be considered malicious interaction. All of the binaries are standalone x86 binaries running in a custom platform named DECREE. DECREE is built on i386 Linux but supports a much smaller set of system calls. So CGC binaries are simpler than CTF challenges, not to mention realistic software.

3.2 Properties of the Dataset

In order to support for vulnerability discovery, exploit generation and other security research, we characterize several properties that are critical and useful. The dataset we are building should cover all of these information as much as possible.

Control flow: Control flow is the order in which individual basic blocks of the target program are executed. It is essential to reflect the root cause of vulnerabilities and demonstrate the workflow of exploits. Software execution must follow a path of target program's Control-Flow Graph (CFG), which is determined ahead of time by source code analysis, binary analysis, or execution profiling. Any violation to the CFG indicates an attack occurred.
There will be many basic blocks executed when running a program. To reduce the load of recording, we only save the entry of a basic block in program sections. As for shared library, we only save the entry and exit addresses of the execution.

Memory maps: A memory map is a structure of data that indicates how memory is laid out. It refers to the mapping between modules of target program, shared library and heap/stack memory regions. The addresses in which the program and libraries are loaded will be randomized if address space layout randomization(ASLR) is enabled. So it is crucial to keep this information in the dataset for future accurate analysis.

Syscall and signal traces: A dataset should also record system calls that are called by a process and signals that are received by the target program. They are very useful for bug isolation, sanity checking and attempting to capture race conditions.

Inputs and Outputs: Inputs is the traffic sent to target program, while outputs is the traffic received from target program. For host-based applications, we could redirect its I/O to network traffic too. The inputs and outputs could be used to replay and for further detail analysis.

Exit status: Exit status indicates whether the process exits normally. By checking the exit status, we can know whether the process crashed. The process will return zero if it exits normally in most operating system. If the process crashed or aborted by signals, the exit status would be different.

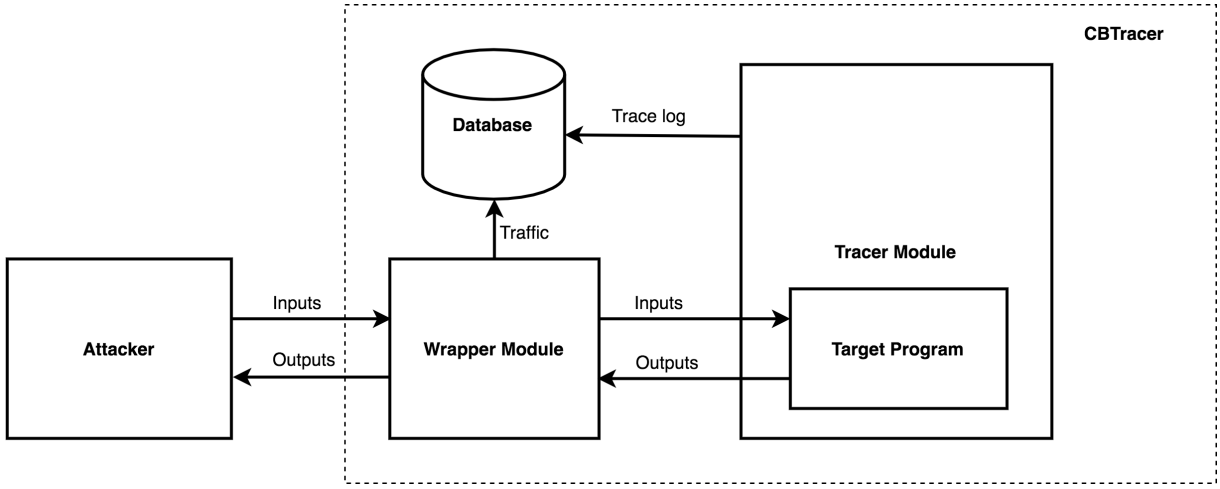


Figure 1: Architecture of CBTracer.

Attackers: Characteristics are used to indicate where the attack comes from. IP addresses are a common feature to identify who the attackers are. A team token, generated for each team uniquely in CTF competitions, can also be used to identify attackers.

Label: The label indicates whether the traffic triggers a vulnerability or not. With this label information, we could provide training dataset for machine learning algorithms.

4 ARCHITECTURE

In order to track the properties mentioned in previous section, we designed and implemented a light weight framework named CBTracer. It dumps information both from the network level and process level.

Figure 1 shows the architecture of CBTracer. It comprises of three core modules: a tracer module for process information dumping, a wrapper module for network traffic dumping, and a database to record information.

Tracer Module. The purpose of tracer module is to save runtime execution contexts when it is executing. It monitors the target process, saves runtime contexts including control flow, memory map and syscall/signal traces.

Wrapper Module. The wrapper module acts as a proxy between attackers and target programs. It receives inputs from attackers and saves the inputs, before sending it to the target program. When outputs are generated from the target program, the wrapper saves the outputs first, and then passes the outputs to the attacker.

A wrapper module can also analyze the traffic. It can detect attacks via third-party solutions such as network intrusion detection systems.

Another job of the wrapper module is to save characteristics of the attacker, including IP addresses and team tokens in CTF competitions.

Database. Database stores all of the data from tracer module and wrapper module. It contains a rich set of information related to executions, e.g., entries of basic blocks, memory maps, inputs, outputs and characteristics of the attacker.

5 IMPLEMENTATION

In this section, we will discuss the implementation of CBTracer. We have implemented CBTracer that supports CTF competitions and CGC binaries. It is a light weight framework that are easy to deploy. The following sections describe the details.

5.1 Wrapper Implementation

When a connection establishes, the wrapper saves the characteristics of the attacker first, including IP address, and a team token in CTF competitions.

Then it will behave as a proxy to dump the inputs and outputs. The following pseudo code shows how the wrapper capture traffic.

```
void wrapper()
{
    char *args[] = {"target_program", NULL};
    int stdin_fds[2];
    int stdout_fds[2];
    pipe(stdin_fds);
    pipe(stdout_fds);
    if (fork())
    {
        createthread(stdin_wrapper_routine);
        createthread(stdout_wrapper_routine);
    }
    else
    {
        dup2(stdin_fds[0], 0);
        dup2(stdout_fds[1], 1);
        execve("target_program", args, NULL);
    }
}
```

It first invokes pipe [28] to create two pipes, one for inputs, the other for outputs. A pipe allocates a pair of file descriptors. The first descriptor connects to the read side of the pipe, the second connects to the write side of the pipe. Then it invokes fork [28]

to create a child process which shares the pipe with the wrapper process. The child process invokes dup2 [28] to duplicate the read side (i.e., inputs for target programs) of the pipe, to file descriptor 0 (i.e. stdin), and duplicates the write side (i.e., inputs for target programs) of the pipe, to file descriptor 1 (i.e., stdout), then invokes execve [28] to execute target program.

The parent wrapper process launched two threads. One sends the received traffic to child process by the pipe. Another sends the traffic received from the pipe to the attacker. We use a buffer to cache traffic in these threads, which will improve data transmission speed. In the meantime, They monitor and save the traffic. As a special hardening solution for CTF competitions, we leverage chroot and seccomp [28] to isolate the file system and limit syscalls which the target process can invoke.

5.2 Tracer Implementation

Instrumentation is a technique which inserts extra code into an application to observe its behavior. The tracer that inserts some extra code into program before a branch instructions can save the start address of the following basic block. Also the tracer could insert some extra code before syscalls to save the contexts of syscalls.

It's easy to implement this type of tracer by using instrumentation tools like PIN [29] with acceptable overhead. But the instrumentation technique is implemented by modifying instructions or using binary translation, which will cause the program to be different from original programs. Although the semantics of the instrumented programs keep the same in general , but its low-level runtime environment (e.g., memory maps) could be different, making some exploits fail.

In CTF game, the challenge runs on a regular Linux system mostly. Changes to memory maps may confuse players and have side effects on exploits, so we didn't implement the tracer module by instrumentation.

Perf [27] is a performance analyzing tool in Linux, available from Linux kernel version 2.6.31. It supports hardware performance counters, tracepoints, software performance counters, and dynamic probes.

Intel Processor Trace (Intel PT) is an extension of Intel Architecture that collects information about software execution such as control flow, execution modes and timings with a low overhead. Linux kernel version 4.1 and later has an integrated PT implementation as part of Linux perf.

CBTracer leverages perf to record control flow, syscall trace. We can also assign a custom perf filter to record various runtime information such as shared library function calls.

Linux provides the ptrace [28] system call for debugging. This system call allows a parent process to observe and control the execution of another process. It can examine and change program's image, data and even values in registers.

CBTracer also supports ptrace to monitor the runtime executions. If Intel PT is not available, CBTracer will trace the target program in single-step mode to record control flow.

It will attach to target programs and stop at every instruction the program executes. When the tracer meets a new step, it fetches address from program counter register and fetches instruction from the address. To trace syscalls, it checks syscall instruction and saves arguments and the return value. If there is a control transfer instruction, the tracer saves the program counter of next step, i.e., the entry of a basic block. If the program counter is transferred into shared library, the tracer only saves the entry and return address of the shared library, ignoring basic blocks in the shared library.

CGC Binaries support ptrace as well. [12] CBTracer can also record runtime executions of CGC binaries via ptrace.

It is worth noting that, the overhead of single-step ptrace is much higher than perf with Intel PT supported.

On Linux, memory maps are stored in `/proc/<pid>/maps`. The tracer saves memory maps by reading this file. Our implementation saves the memory map when the program starts and exits.

6 EVALUATION

6.1 Performance

Deployed with CGC binaries, CBTracer runs locally to get the data. In such scenario, we should not be overly concerned about the performance. In online CTF competitions, A good performance has less influence on players and saves the computing resource.

The cost of time to exploit a challenge is directly perceived through the timers. To evaluate the performance in CTF competitions, we deployed CBTracer with five challenges of BCTF2017. We collected players' exploits from their writeups, then we exploited these challenges with different implementation and recorded the average runtime of 1000 exploits. Figure 2 shows the results.

It takes some time to launch tracer with perf. In spite of this, the overhead of the tracer with Intel PT is little. There are many interactions in `100levels` and we use buffer when implementing the wrapper, so the cost of tracer with Intel PT is slightly less than original deployment in `100levels`.

Tracing via ptrace takes much more time to run a successful exploit on average. While tracing via perf with Intel PT supported only has little overhead.

6.2 Side Effects on Exploits

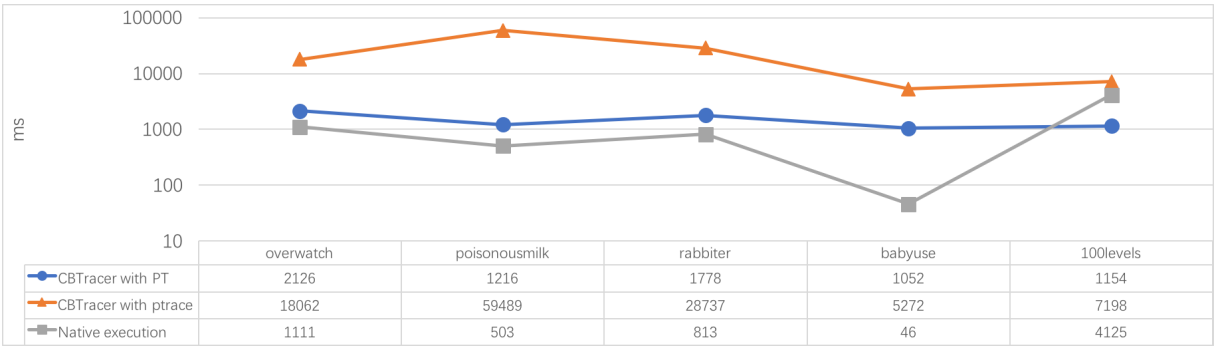
Figure 3 shows the results that we exploited these challenges locally with different implementation. Some exploits are not so stable that can't capture the flag every time.

The tracer doesn't insert any instruction to the attached process except interrupt instructions, so it won't have effect on the exploit. The result proves it. The success rates of exploit in challenges with different deployment are approximate. According to the result, we think that our implementation has hardly any effect on exploits in these challenges.

7 BUILDING DATASET

Building the dataset is a work in progress. We have collect data from CGC binaries and CTF competitions.

CGC binaries works in DECREE which is incompatible with some tools on Linux. So we leveraged cb-multios [51] to run CGC binaries on Linux. We have run PoVs and polls on these binaries with CBTracer to get various data. Some of our work needs dataset which contains CGC binaries with ASLR enabled, so we compiled the CGC binaries with PIE enable and collected the data. We labeled PoV-generated traffic as malicious traffic, and labeled poll-generated traffic as benign traffic.



	overwatch	poisonousmilk	rabbiter	babyuse	100levels
CBTracer with PT	2126	1216	1778	1052	1154
CBTracer with ptrace	18062	59489	28737	5272	7198
Native execution	1111	503	813	46	4125

Figure 2: Time consumed of 1000 exploits in each challenge with different modules deployed.

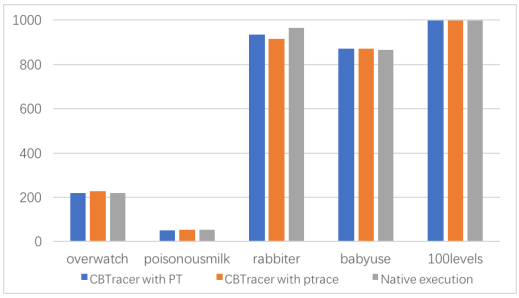


Figure 3: Number of successful exploit of 1000 attempts in each challenge with different modules deployed.

We deployed our system in BCTF 2017 [9]. We could only use ptrace to record control flow which took a large overhead because AWS doesn't support Intel PT. Learning from lessons learned, we are going to deploy CBTracer with Intel PT enabled in the following CTF competitions we hold.

In attack-defense CTF, We labeled availability-check traffic as benign traffic and labeled other traffic as malicious traffic. It is a challenge to identify a successful exploit. We discuss some solutions below.

CFI. The CFI [1] security policy states that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. The CFG in question can be defined by source code analysis, binary analysis, or execution profiling. Any violation of CFG could be used to detect the exploitation which hijacks the control flow.

Exit Status Checking. The exit status of a process indicates whether the process exits normally. The process will return zero if it exits normally in most operating system. While the program will exit with different exit status if it exits abnormally or receives a signal.

Flag Detection. In CTF competitions, a flag (i.e., a specific string in the program or in the system) is the proof of solving the challenge. Teams that successfully exploit the challenges could usually gain access to the flag memory or files in the system. Detecting flag in traffic or detecting operation on flag file can be used to define a successful exploit.

Real-world software is quite different with CTF and CGC challenges. We are improving CBTracer to support some real-world software. We have to custom wrapper for every software, because the ways to interact with them are different. Multithreading and Multiprocessing are widely used in real-world software. The tracer module have to be improved to support it.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we discussed the properties of the dataset and some possible source of data. We designed and implemented CBTracer to continuously build datasets for security analysis from CGC and CTF challenges. The experiment results shows that CBTracer can collect security-related data effectively with almost no effect on players. In the future, we will improve CBTracer to support real-world software and build dataset from it.

The dataset built from CGC and CTF challenges can be used in much research, and we call for contributors to deploy CBTracer in their CTF competitions to make the dataset bigger. CBTracer is a light weight framework which can be deployed with CTF challenges easily. Besides, we organize XCTF league and some international CTF competitions such as BCTF. We believe that we will collect more data from these CTF competitions to build a rich dataset.

9 ACKNOWLEDGEMENTS

Our paper is supported by the National Natural Science Foundation of China(Grant No.61472209) and Tsinghua University Initiative Scientific Research Program(Grant No.20151080436).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[3] Piotr Bania. 2009. Evading network-level emulation. *arXiv preprint arXiv:0906.1963* (2009).

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 380–394.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[7] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture* 57, 3 (2011), 294–313.

[8] Jedidiah R Crandall, Zhendong Su, S Felix Wu, and Frederic T Chong. 2005. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 235–248.

[9] CTFtime. [n. d.]. BCTF 2017. https://ctftime.org/event/432. ([n. d.]).

[10] Marc Dacier, Corrado Leita, Olivier Thonnard, Hau Van Pham, and Engin Kirda. 2010. Assessing cybercrime through the eyes of the WOMBAT. In *Cyber Situational Awareness*. Springer, 103–136.

[11] DAPRA. [n. d.]. DARPA Cyber Grand Challenge. http://www.cybergrandchallenge.com. ([n. d.]).

[12] DARPA. [n. d.]. ptrace for DECREE. https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/ptrace-for-decree.md. ([n. d.]).

[13] Yu Ding, Tao Wei, Hui Xue, Yulong Zhang, Chao Zhang, and Xinhui Han. 2017. Accurate and efficient exploit capture and classification. *Science China Information Sciences* 60, 5 (2017), 052110.

[14] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.

[15] Michael Eddington. 2011. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.

[16] Fireeye. [n. d.]. Fireeye Malware Analysis Sandbox. https://www.fireeye.com/products/malware-analysis.html. ([n. d.]).

[17] Prahlad Fogla, Monirul I Sharif, Roberto Perdisci, Oleg M Kolesnikov, and Wenke Lee. 2006. Polymorphic Blending Attacks.. In *USENIX Security*.

[18] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

[19] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. *arXiv preprint arXiv:1701.07232* (2017).

[20] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward large-scale vulnerability discovery using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 85–96.

[21] Ashlesha Joshi, Samuel T King, George W Dunlap, and Peter M Chen. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 91–104.

[22] Paul Jung. 2014. Bypassing Sandboxes for Fun. (2014).

[23] Neal Krawetz. 2004. Anti-honeypot technology. *IEEE Security & Privacy* 2, 1 (2004), 76–79.

[24] Christian Kreibich and Jon Crowcroft. 2004. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM computer communication review* 34, 1 (2004), 51–56.

[25] Ted Kremenek. 2008. Finding software bugs with the clang static analyzer. *Apple Inc* (2008).

[26] Zhichun Li, Lanjia Wang, Yan Chen, and Zhi Fu. 2007. Network-based and attack-resilient length signature generation for zero-day polymorphic worms. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*. IEEE, 164–173.

[27] Linux. [n. d.]. Linux perf. https://perf.wiki.kernel.org. ([n. d.]).

[28] Linux man page [n. d.]. Linux man page. https://linux.die.net/man. ([n. d.]).

[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[30] Jose Nazario. 2009. PhoneyC: A Virtual Client Honeypot. *LEET* 9 (2009), 911–919.

[31] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 529–540.

[32] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. (2005).

[33] Thuy TT Nguyen and Grenville Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials* 10, 4 (2008), 56–76.

[34] Viet Hung Nguyen and Le Minh Sang Tran. 2010. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*. ACM, 3.

[35] NIST. [n. d.]. National Vulnerability Database. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd. ([n. d.]).

[36] Offensive Security. [n. d.]. Exploits Database. https://www.exploit-db.com/. ([n. d.]).

[37] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23 (1999), 2435–2463.

[38] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 426–437.

[39] Niels Provos. 2003. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, Vol. 2. 4.

[40] Niels Provos et al. 2004. A Virtual Honeypot Framework.. In *USENIX Security Symposium*, Vol. 173. 1–14.

[41] Rapid7. [n. d.]. Metasploit: Penetration Testing Software. https://www.metasploit.com/. ([n. d.]).

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. (2017).

[43] NIST SAMATE. [n. d.]. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/testsuite.php. ([n. d.]).

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[45] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision.. In *USENIX Annual Technical Conference, General Track*. 17–30.

[46] Yonghee Shin and Laurie Williams. 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 315–317.

[47] Yonghee Shin and Laurie Williams. 2008. Is complexity really the enemy of software security?. In *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 47–50.

[48] Kuldeep Singh and Sunil Agrawal. 2011. Comparative analysis of five machine learning algorithms for IP traffic classification. In *Emerging Trends in Networks and Computer Communications (ETNCC), 2011 International Conference on*. IEEE, 33–38.

[49] Jungsuk Song, Hiroki Takakura, Yasuo Okabe, Masashi Eto, Daisuke Inoue, and Koji Nakao. 2011. Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM, 29–36.

[50] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*.

[51] trailofbits. [n. d.]. cb-multios. https://github.com/trailofbits/cb-multios. ([n. d.]).

[52] Matthew Van Gundy, Davide Balzarotti, and Giovanni Vigna. 2007. Catch Me, If You Can: Evading Network Signatures with Web-based Polymorphic Worms.. In *WOOT*.

[53] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. 2000. ITS4: A static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 257–267.

[54] Ryan Whelan, Tim Leek, and David Kaeli. 2013. Architecture-independent dynamic information flow tracking. In *International Conference on Compiler Construction*. Springer, 144–163.

[55] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 41–50.

[56] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*. USENIX Association, 13–13.

[57] Michal Zalewski. 2007. American fuzzy lop. (2007).

[58] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. 2012. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis. (2012).

[59] Cliff Changchun Zou and Ryan Cunningham. 2006. Honeypot-aware advanced botnet construction and maintenance. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE, 199–208.