Dataflow Analysis for Known Vulnerability Prevention System[i]

Lifang Qin[1], Yichao Li[2], Cao Yue[3]
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Chengdu, China
lifang_qin@sina.com.cn[1] richardlyc@uestc.edu.cn[2] yuecao@uestc.edu.cn[3]

***Abstract*—Usually, a new exploit for a single vulnerability can readily be turned into worms which compromise hundreds of thousands of machines within only a few minutes. In order to protect the host from malicious attacks, we propose a new approach for automatic defense mechanism: Dataflow Analysis for Known Vulnerability Prevention System (shortly for DA-VPS), which has properties with easy deployment, accurate detection and low overhead. In our paper, we present the principle, architecture, implement, and deployment of DA-VPS.**

***Keywords*—vulnerability, dataflow analysis, malicious attack detection**

I. INTRODUCTION

With the rapidly developing of computer technology, the security of the host system hasn't been strengthened. At most, it just compensates the increasing degree of security threat as the result of the complexity of computer application environment. Moreover, it is possible that the appearance of new technologies will make the security of computer system more brittle.

Via theory analysis, the main reason why attack behaviors, such as computer virus, malicious code, and network invade, can bring great threat to the computer system is that there exists many vulnerabilities during the course of designing, developing, and maintaining system or application software. A new exploit for a single vulnerability can readily be turned into worms which compromise hundreds of thousands of machines within only a few minutes. Hence, a good defense system should take a rapid response after new vulnerability discovered and repair the vulnerability before attackers exploit it.

At present, both scholar and organization are doing great research on decreasing, detecting and defending vulnerability. They have proposed several approaches to detect vulnerabilities when a program is exploited. Most of these previous mechanisms, such as StackGuard [1], PointGuard [2], full-bounds check [3], LibsafePlus [4], FormatGuard [5], and CCured [6], require source code or special recompilation of the program. Some of them also require recompiling the libraries [3], or modifying the original source code, or are not compatible with some programs [2, 6]. These constraints hinder the deployment and applicability of these methods, especially for commodity software whose source code are often unavailable, and the additional work required (such as recompiling the libraries and modifying the original source code) makes it inconvenient to apply these methods to a broad range of applications. Note that most of the large-scale worm attacks to date are attacks on commodity software.

Thus, a good defense system needs to meet several goals simultaneously [7]:

1) Fast defense development and deployment. There is often very little reaction time, especially when the exploit comes in the form of a fast propagating worm.

2) No requirement for source code. Many vulnerable programs are commodity software for which the source code is proprietary. To respond quickly to new vulnerabilities, we need to develop a defense mechanism without access to source code.

3) High accuracy and effectiveness. The defense mechanism should protect against the vulnerability and should not have any undesirable side effect on normal execution.

4) Low performance overhead. The defense mechanism should have low performance overhead, so a vulnerable host deploying the defense mechanism can still provide critical services with little performance degradation.

In this paper, we propose Dataflow Analysis for Known Vulnerability Prevention System (shortly for DA-VPS): a new approach for automatic defense which has properties with easy deployment, accurate detection and low overhead. DA-VPS can protect the target system effectively and ensure the target program in its normal execution while it is in danger of malicious attacks.

II. APPROACH: DATAFLOW ANALYSIS FOR KNOWN VULNERABILITY PREVENTION SYSTEM

In our paper, we propose a new defense mechanism: Dataflow Analysis for known Vulnerability Prevention System (shortly for DA-VPS), which is implemented on DynamoRIO [8].

The principle of our DA-VPS is: firstly, using dataflow analysis technology, DA-VPS traces the data coming from the untrusted source of input, and diagnoses whether it is the tainted. Secondly, DA-VPS examines the spread path of tainted in a backward manner to determine which instructions propagate the tainted and log them. Thirdly, according to vulnerability context and the spread path of tainted, DA-VPS generates the vulnerability filter and its hot patch. Before the next exploit reaches the exploit point, using the vulnerability

filter, DA-VPS can detect the exploit and then applies the related hot patch to repair this vulnerability.

Figure 1 shows the overall architecture of DA-VPS. Our architecture contains four components: TaintedTracker, Vulnerability Analyzer, Hotpatch Generator and Vulnerability Repair.

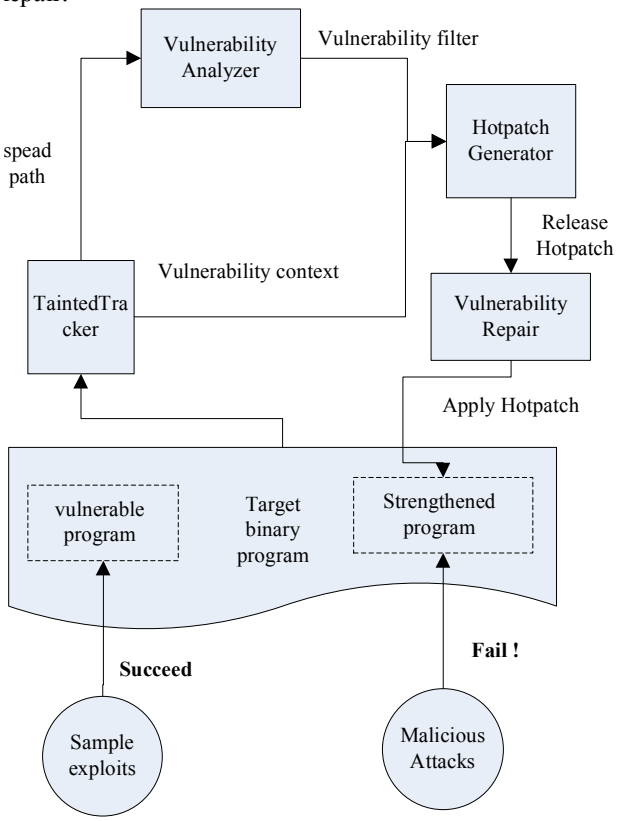

Figure 1. The arcchitecture of DA-VPS

In Figure 1, we assume the sample exploits address to the known vulnerabilities.

III. DESIGN AND IMPLEMENT

In this part, we illustrate the implement of our DA-VPS.

A. TaintedTracker

Usually, an attacker is trying to change the execution of a program illegitimately. So he must cause a value that is normally derived from a trusted source to instead be derived from his own input. For example, values such as jump addresses and format strings should usually be supplied by the code itself, not from external untrusted inputs. However, an attacker may attempt to exploit a program by overwriting these values with his own data.

As the first part of DA-VPS, TaintedTracker is a new detector, which can detect attacks such as buffer overflow or format strings exploit. In our paper we refer to data that originates or is derived arithmetically from an untrusted input as being tainted. The responsibility of TaintedTracker is to mark input data derived from untrusted sources as tainted, and then monitor the execution of program, which can trace how the tainted attribute propagates (i.e., what other data becomes tainted). TaintedTracker also checks when tainted data is used in dangerous ways. For example, using tainted data as jump addresses or format strings often indicates an exploit of a vulnerability such as a buffer overrun or format string vulnerability.

TaintedTracker performs dataflow analysis on a program by running the program in its own emulation environment. Specially, we implement DA-VPS Using DynamoRIO [8, 9], a dynamic binary instrumentation tool. Actually, James Newsome and Dawn Song referred the dataflow analysis as dynamic taint analysis [10]. Similar to their TaintCheck, our TaintedTracker is consisted of three parts: TaintMarking, TaintTracing, and TaintAssert.

1) TaintMarking

TaintMarking marks any data that comes from an untrusted source of input as tainted. Here, an untrusted source is the input which may bring any attacker's data. By default, TaintMarking considers input from network sockets to be untrusted, since for most programs the network is the most likely vector of attack. By an extended policy, other sources inputs e.g. input data from certain files or stdin can be considered as untrusted sources.

Taint Marking uses shadow memory to mark tainted. Shadow memory is a memory mapping technology which maps one memory address to another. Each byte of memory, including the registers, stack, heap, etc., has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, otherwise a NULL pointer if it is not. We use a page-table-like structure to ensure that the shadow memory uses very little memory in practice. TaintMarking examines the arguments and results of each system call, and determines whether any memory written by the system call should be marked as tainted or untainted according to the policy. When the memory is tainted, TaintMarking allocates a Taint data structure that records the system call number, a snapshot of the current stack, and a copy of the data that was written. The shadow memory location is then set to a pointer to this structure. This information can later be used by the VulnerabilityAnalyzer when an attack is detected. Optionally, logging can be disabled, and the shadow memory locations can simply store a single bit indicating whether the corresponding memory is tainted.

2) TaintTracing

Because other data in the memory may be tainted when manipulating tainted data, we need to trace how the tainted propagates in the program. Therefore, TaintTracing will track each instruction that manipulates data in order to determine whether the result is tainted. The type of Instructions traced by TaintTracing mainly includes data movement instructions (like LOAD, STORE, MOVE, PUSH, POP, etc.) and arithmetic instructions (ADD, SUB, XOR, etc.). For data movement instructions, the data at the destination will be tainted if and only if any byte of the data at the source location is tainted. For arithmetic instructions, the result will be tainted if and only if any byte of the operands is tainted. While arithmetic instructions also affect the processor's condition flags, we do not track whether the flags are tainted, because it is normal for

untrusted data to influence them. Note that for both data movement and arithmetic instructions, literal values are considered untainted, since they originate either from the source code of the program or from the compiler.

In order to track the propagation of tainted data, TaintTracing adds instrumentation before each data movement or arithmetic instruction. When the result of an instruction is tainted by one of the operands, TaintTracing sets the shadow memory of the result to point to the same taint structure as the tainted operand. Optionally, TaintTracing can instead allocate a new taint structure with information about the relevant instruction including the operand locations and values, and a snapshot of the stack, which points back to the previous taint structure. When an attack is detected, the VulnerabilityAnalyzer can follow this chain of Taint structures backwards to determine how the tainted data propagated through memory.

3) TaintAssert

By marking data as tainted correctly and tracking its propagation at run time, we can detect malicious behavior. TaintAssert checks whether tainted data is used in ways that its policy defines as illegitimate. TaintAssert's default policy is designed to detect format string attacks, and attacks that alter jump targets including return addresses, function pointers, or function pointer offsets. When Taint Tracker detects that tainted data has been used in an illegitimate way, signaling a likely attack, it invokes the Vulnerability Analyzer to further analyze the attack.

TaintAssert checks Jump addresses, format strings, and system call argument. By default, TaintAssert checks whether tainted data is used as a jump target (such as a return address, function pointer, or function pointer offset) or as a format string argument to the print family of standard library functions. Many attacks attempt to overwrite one of these in order to redirect control flow either to the attacker's code, to a standard library function, such as exec, or to another point in the program (possibly circumventing security checks). Hence, firstly, for Jump addresses, by having Taint Tracker place instrumentation before each jump instruction we can ensure that the data specifying the jump target is not tainted. Secondly, for format strings, TaintAssert can check whenever tainted data is used as a format string, even if it does not contain malicious format specified for attacks. This could be used to discover previously unknown format string vulnerabilities. Thirdly not the lastly, for system call argument, TaintAssert can check whether particular arguments to particular system calls are tainted, though this is not enabled in TaintAssert's default policy. This could be used to detect attacks that overwrite data that is later used as an argument to a system call.

B. Vulnerability Analyzer

When tainted data is asserted to be misused, TaintTracker provides some useful information about how the exploit happened, and what the exploit attempts to do. These are very help for identifying vulnerabilities and for generating exploit signatures. And then we call Vulnerability Analyzer to make further analysis.

Information logged by TaintTracker shows the relevant part of the execution path in between tainted data's entry into the system, and its use in an exploit. By back-tracing the chain of Taint structures, the Vulnerability Analyzer provides information including the original input buffer that the tainted data came from, the program counter and call stack at every point the program operated on the relevant tainted data, and at what point the exploit actually occurred. The Vulnerability Analyzer can use this information to help determine the nature and location of a vulnerability quickly, and to identify the exploit being used.

Optionally, the Vulnerability Analyzer can allow an attack to continue in a constrained environment after it is detected. We currently implement an option to redirect all outgoing connections to a logging process. This could be used to collect additional samples of a worm, which can be used to help generate a signature for that worm.

C. Hotpatch Generator

Hotpatch Generator component includes two type instructions: tainted spread path instructions and tainted data misused instructions. Additionally, tainted spread path instructions contain data movement instructions and arithmetic instructions. Generator gets the execution path information of exploit for specified vulnerability from TaintTracker, traces in a backward manner from the exploit point, and generates the Hot-Patch.

Hotpatch is the regulation for discovering, filtering, and repairing the vulnerability. Hotpatch Generator changes all instructions of the Hotpatch into memory space addresses, which are used to monitor the execution of the program.

D. Vulnerability Repair

Vulnerability Repair monitors the running instructions and memory address related to the Hotpatch. When the terminal defense system receives the Hotpatch sent from remote services center, Vulnerability Repair detects whether there exists the specified vulnerability in the local system. If there exists, it applies the Hotpatch and repairs the vulnerability.

Since the Hotpatch is the form of instructions, the instruction list and the instruction position to exploit point, Vulnerability Repair can detects the exploit once it appears. And before attack reaches the exploit point, Vulnerability Repair applies the method associated to Hotpatch to rewrite the code-flow of program so that attacker fails the exploit in a new environment.

IV. DEPLOYMENT

Our DA-VPS has satisfied three important goals: Fast vulnerability filter generator, accurate detection, and low performance overhead, which ensure our DA-VPS can protect target host from the threat of network attacks.

Figure 2 shows the deployment of DA-VPS.

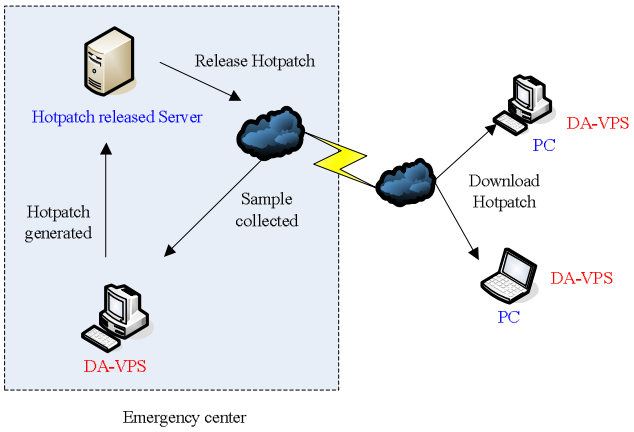

Figure 2. *The deployment of DA-VPS*

Terminal net hosts have installed DynamoRIO and our DA-VPS system.

Through Honeypot or vulnerability release website, Emergency center can collect the sample exploits for latest vulnerability of program, and then submit them to Vulnerability Analysis.

DA-VPS can generate vulnerability filter, and issues Hotpatch to terminal users.

V. CONCLUSION

In this paper, we point out the limitations of current detection mechanism. Then we propose a new approach for automatic defense: Dataflow Analysis for Known Vulnerability Prevention System (DA-VPS), which has a rapid reaction when a new vulnerability is discovered. We give its overall architecture in part 2 and describe the detail of its implement in part 3. The deployment of DA-VPS is also illustrated in part 4. To sum up, DA-VPS can protect the target system effectively and ensure the target program in its normal execution while it is in danger of malicious attacks.

[1] Crispin Cowan, Calton Pu, Dave Maier, JonathonWalpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, January 1998.

[2] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In 12th USENIX Security Symposium, 2003.

[3] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In Proceedings of the 11th Annual Network and Distributed System Security Symposium, February 2004.

[4] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In USENIX Security Symposium, August 2004.

[5] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg KroahHartman. FormatGuard: automatic protection from printf format string vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.

[6] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In Proceedings of the Symposium on Principles of Programming Languages, 2002.

[7] James Newsome, David Brumley, Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. NDSS, 2006.

[8] Derek L. Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Electrical Engineering and Computer Science. September, 2004.

[9] Dynamorio. http://www.cag.lcs.mit.edu/dynamorio/.

[10] James Newsome, Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. NDSS. May,2004.

[11] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, Yuanyuan Zhou, James Newsome, David Brumley, Dawn Song. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. Proceedings of the 2nd ACM SIGOPS EuroSys (EuroSys'07). March, 2007.

[12] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability Driven Network Filters for Preventing Known Vulnerability Exploits. In the Proceedings of ACM SIGCOMM, Portland. August,2004.

i This research is partially supported by the National Information Security 242 Program of China under Grant No. 2007B30