Hybrid Defense: How To Protect Yourself From Polymorphic 0-days

Svetlana Gaivoronski and Dennis Gamayunov

Information Systems Security lab, CS Dept. at Lomonosov Moscow State University
Russia, 119899 Moscow, Leninskie Gory 1, Moscow State University, second EDU building, Computer Science Dept., room 603
{sadie,gamajun}@lvk.cs.msu.su

1 Introduction

The problem of shellcode detection still challenges researchers in industry, academia, CTF white and gray teams, since it is a significant part of the more general problem of targeted or blind automatic attack detection and filtering. The total amount of remotely exploitable vulnerabilities found in networking software every year continues to grow despite of the significant effort in the software industry to improve the code quality. From that point of view, it's worth to remember the latest vulnerabilities, for example, described at MS12-020 report, and for latest 0-days, such as client-side 0-day in Java revealed in late August 2012. Also, memory corruption errors are often used as a part of CTF tasks, and successul exploitation usually gives it all to the winner - flags, applications, systems, position in the contest table, etc. Having a fast and accurate shellcode detector would help not only to improve defense skills and level, but also to gather new ideas from other teams, who try to crack your boxes all the game long. And, in fact, any fast and accurace shellcode detector would help you improve your opensource or commercial defensive software, or maybe spin off one more of them.

But when we try to detect 0-day attacks or network worm propagation at network level, we come across hardware processing power limitation. At this level we observe an evolving gap between computational power of the modern processors and the throughput of the network channel. Therefore, we should pay special attention to the computational complexity of the algorithms used for malware propagation detection. Also, false positives rate of the algoritms is very important as it may render any tool unusable even if it has zero false negatives rates.

So, we tried to bring a little science to this tiny problem, and construct a hybrid shellcode classifier, which is actually an automatic builder of hybrid shellcode classifiers, each of them may be optimal in terms of speed and FP rates in different cases. For example, in CTF setup we would expect lots of malicous traffic going over the cable, and in some peaceful university campus on the contrary we would see terabytes of "benign" multimedia data. In these two situations two different architectures may be optimal, and we're presenting the tool which helps to build them automatically.

2 Existing approaches and their limitations

.

According to the principles at work, shellcode detection methods can be divided into the following classes:

– *static methods* - methods of code analysis without executing it;
– *abstract execution* - analysis of code modifications and accessibility of certain blocks of the code without a real execution. The analysis uses assumptions on the ranges of input data and variables that can affect the flow of execution;
– *dynamic methods* - methods that analyze the code during it's execution;
– *hybrid methods* - methods that use a combination of static and dynamic analysis and the method of abstract interpretation.

From a theoretical point of view, static analysis can completely cover the entire code of the program and consider all possible objects S, generated from the input stream. In addition, static analysis is usually faster than dynamic. Nevertheless, it has several shortcomings:

– A large number of tasks which rely on the program's behavior and properties, can't be solved by using static analysis in general:
  - Problem of detecting metamorphic shellcode by static analysis is undecidable.
  - The problem of detection of polymorphic shellcode is NP-complete in the general case.
– The attacker has the ability to create malicious code which is static analysis resistant. In particular, one can use various techniques of code obfuscation, indirect addressing, self-modifying code techniques, etc.

In contrast to static methods, dynamic methods are resistant to the code obfuscation and to the various anti-static analysis techniques (including self-modification). Nevertheless, the dynamic methods also have several shortcomings:

– they require much more overheads than static analysis methods. In particular, a sufficiently long chain of instructions can be required to conclude whether the program has malicious behavior or not;
– the coverage of the program is not complete: the dynamic methods consider only a few possible variants of program execution. Moreover, many significant variants of program execution can not be detected;
– the environment emulation in which the program exhibits it's malicious behavior is difficult;
– there are detection techniques for program execution in a virtual environment. In this case, the program has the ability to change it's behavior in order not to exhibit the malicious properties.

During consideration of existing shellcode detection methods, we concludes, that methods with low computational complexity almost always demonstrate

high false positives rates. These methods allow to process large volumes of data in real time,but have too high false positives rates to be practical. Methods with low false positives rate typically have high computational complexity, which makes them useless for network traffic analysis.

3 Shellcode features and classes

. During our work, we identified shellcode features which are used to distinguish shellcodes from benign data. None of the features may be used as precise shellcode marker, but their presense definitely increase probability for the given sample of being real working shellcode.

Features can be generic or specific: generic features correspond to all existing shellcodes, and specific features correspond to certain shellcode *family*. Moreover, features can be divided into static and dynamic: static features can be detected by static analysis, and dynamic features only appear during code execution.

Static features

- Correct disassembly into chain at least K instruction;
- Number of push-call patterns exceeds threshold;
- Overall shellcode size does not exceed threshold;
- Operands of self-modifying and indirect jmp are initialized;
- Cleared IFG contains chain with more than N instructions;
- Correct disassembly from each and every offset. This feature is specific fshellcode which contain NOP-sled;
- The value of maximum execution length (number of instructions) exceeds threshold. Feature is also specific for NOP-sled containing shellcodes;
- Conditional jumps to the lower address offset. Feature is specific for encrypted shellcodes;
- Return address lies within certain range of values. Feature is specific for non-ASLR systems;
- Presence of GetPC. Specific for encrypted shellcodes;
- Last instruction in the chain ends with branch instruction with immediate or absolute addressing targeting lib call or valid interruption. Specific for non-ASLR systems;

Dynamic features

- Number of near reads within payload exceed threshold R;
- Number of unique writes to different memory location exceeds threshold W;
- Control at least once transferred from executed payload to previously written address. Feature is specific for non-self-contained shellcode - shellcode which doesn't involve any kind of GetPC code and doesn't read its own memory address during decryption;
- Execution of wx-instruction (write-execute) exceeds threshold X. Feature is specific for non-self-contained shellcodes;

Any modern shellcode is a combination of certain shellcode features and benign code features. For example, long sequence of `0x90` instruction was often used in proof-of-concept exploits, but it can be found in benign data and executables quite often. Using unique shellcode features and benign executables' features, shellcode space may be divided into "families" - a set of shellcode examples that based on similar features and similar execution patterns. In this paper we propose a list of those "families", or shellcode classes. Shellcodes are divided into classes depending on what part of shellcode they represent - Activator, Decryptor, Payload, or Return address zone.

4 Proposed approach

Many of existing methods use the same basic steps in data analysis: disassembly stage, reconstruction of CFG,IFG, etc. Thus,it seems promising and feasible to develop the shellcode detection library in a form of simple set of elementary classifiers - detectors of shellcode features. Using such set of elementary classifiers we formulate the problem of automatic synthesis of hybrid shellcode detector which would cover all shellcode classes and reduce the false positives rate while reducing the computational complexity of the method compared with the simple linear combination of classifiers. We consider the problem of algorithm synthesis as construction of a directed graph $G(V;E)$ with a specific topology, where $\{V\}$ is the set of nodes which are classifiers themselves, $\{E\}$ is the set of arcs. Each arc represents the route of data flow. We assume that if some classifier concludes sample to be legitimate, the sample is not passed to other classifiers. If classifier concludes sample to be malicious, the sample is redirected to the next level of the graph. This redirection is a "double-check" of previous classifier result, and it helps reducing the total false positives rate.
The hybrid classifier should satisfies the following requirements:

- the upper level of graph should provide full coverage of those shellcode classes, which can be detected by available elementary classifiers;
- the higher the level position, the more optimal in terms of time complexity classifiers' set.

This leads to the fact that legitimate flow will no pass the upper levels of the graph. Thus, classifiers with higher computation complexity, which are located at the lower levels will not be executed. Given the fact that malware percentage compared with legitimate data flow in real channels is rather small, this leads to significant decrease of computational complexity of the hybrid classifier.
Example of hybrid classifier is presented in figure 1.

5 Evaluation

A prototype of shellcode detection library and of the proposed method was implemented and evaluated for different data sets (exploits generated by Metasploit

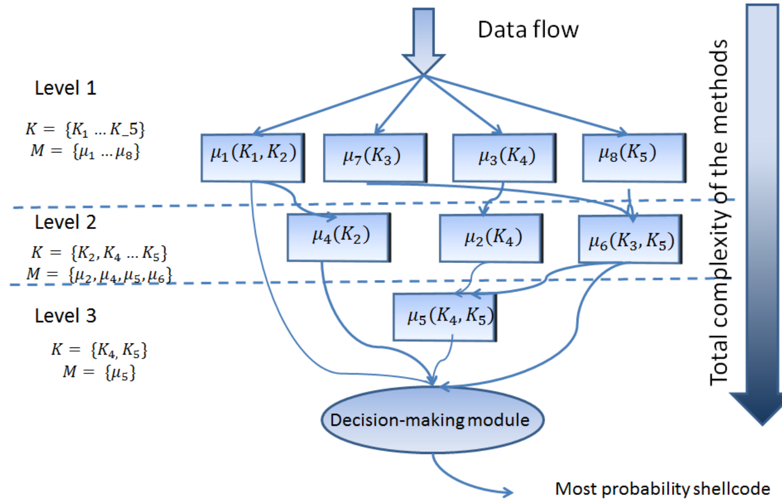

Fig. 1. Hybrid shellcode detection graph. $\mu_{[1-5]}$ stands for elementary classifier, $K_{[1-7]}$ stands for shellcode classes.

- 1536 items, legitimate executables from /usr/bin/ and Windows binaries - 2000 items, random data - 100 Mb, multimedia - 100 Mb). The prototype implementation was tested in Virtual Machine (with Ubuntu 10.1) on machine with the following characteristics: Intel Core 2 Duo CPU, 2.53 HGz, 4 GB RAM.

One of the important goals of our work is to minimize false positives rates when detecting shellcode. The minimum value of false positives rate may be achieved with a simple linear combination of existing classifiers. In such linear topology data flow passes through *all* classifiers and after that decision-making module analyzes their output.

In our experiments we compare the proposed hybrid approach with a linear graph topology. The results for false positives rate, false negatives rate and for average throughput for linear and hybrid topologies are shown in table 1.

	Linear			Hybrid		
Data set	fn, ·100%	fp, ·100%	v, Mb/sec	fn, ·100%	fp, ·100%	v, Mb/sec
exploits	0.2	n/a	0.069	0.2	n/a	0.11
leg. binaries	n/a	0.0064	0.15	n/a	0.019	2.36
random data	n/a	0	0.11	n/a	0	3.7
multimedia	n/a	0.005	0.08	n/a	0.04	3.62

Table 1. Comparison of testing results for linear and hybrid topology. fn stands for false negatives rate, fp - for false positives rate, v - for average throughput.

Thus, hybrid data-flow classifier significantly boosts analysis throughput for benign data - up to 45 times faster than linear combination of classifiers, and almost 1.5 times faster for shellcode only datasets. The absolute throughput value of the current implementation - about 4MB/s (32Mbit/s) - is still to low for "blind" traffic analysis, disregarding application protocols semantics. But in combination with data sampling techniques, parallel processing and content-aware input filtering the hybrid shellcode classifier may be useful for 0-day detection and filtering even in 1Gbps environment.

Implementation of the hybrid classifier used for evaluation described in this paper is available for research community for further development and experimentation at https://gitorious.org/demorpheus.

6 About the authors

Svetlana Gaivoronski is a PhD student at Computer Systems Lab, Computer Science Dept. of Moscow State University, Russia. Svetlana is a member of the Bushwhackers CTF team which shows the following results in recent years: 2nd place in Deutsche Post Security Cup 2010, 6th place in the final of ruCTF 2012 (8th at qualification), 12th place at ruCTF Europe 2011, 4th place in the final of ruCTF 2011 (and 1st at qualification), etc. Svetlana works at Redsecure project (experimental IDS/IPS) at Moscow State University. Her primary interests are network worm propagation detection and filtering, shellcode detection, static and runtime analysis of malware.

Dennis Gamayunov is a PhD, Senior Researcher, and Acting Head of the Information Systems Security Lab, Computer Science Dept. of Moscow State University, Russia. He is also the leader of the network security research group in MSU, project lead of the experimental event-drive and natively multicore Redsecure IDS/IPS. Dennis is the founder of Bushwhackers CTF team, with primary research and practical interests in network level malcode detection, high-speed traffic processing (including FPGA-based), and OS security with fine-grained privilege separation, SELinux and beyond.