Journal of Information Security, 2014, 5, 37-45
Published Online April 2014 in SciRes. http://www.scirp.org/journal/jis
http://dx.doi.org/10.4236/jis.2014.52004

Detection of Sophisticated Network Enabled Threats via a Novel Micro-Proxy Architecture

Andrew Blyth

Information Security Research Group, University of South Wales, Pontypridd, UK
Email: andrew.blyth@southwales.ac.uk

Received 8 January 2014; revised 8 February 2014; accepted 15 February 2014

Abstract

With the increasing use of novel exploitation techniques in modern malicious software it can be argued that current intrusion detection and intrusion prevention systems are failing to keep pace. While some intrusion prevention systems have the capability to detect evasion techniques they all fail to detect novel unknown exploitation techniques. Traditional proxy approaches have failed to protect the universe of discourse that a network enabled service can be engaged in as they view all information flows of the same type in a uniform manner. In this paper we propose a micro-proxy architecture that utilizes reverse engineering techniques to identify a valid universe of discourse for a network service. This valid universe of discourse is then applied to validate legitimate transactions to a service. Thus in effect, the micro proxy implements a default deny policy via the analysis of the application level discourse.

Keywords

Network Security; Firewalls; Proxies and Intrusion Detection

1. Introduction

Organizations today seek to utilize information technology as part of their business processes. Networks allow for organizations to communicate with other organizations and people across the globe in real-time. This allows organizations to integrate their supply chains and perform just in time supply-chain management. However this level of connectivity and dependency on information technology can be viewed as a risk as it has the potential to allow an adversary to access confidential/sensitive information and act upon it. Thus organizations have a requirement to protect and defend their information technology so as to maximize their business utility. Over the past three years the following high profile sophisticated computer network attacks have functioned to illustrate how Computer Network Attack (CNA) has moved into the area of information acquisition and intelligence gathering via the application of zero-day exploits. This indicates that the nature and capabilities of threats and threat agents continues to evolve.

- **Ghostnet:** This the name given to the cyber spying operation discovered in March 2009. It infiltrated high-value political, economic and media locations in 103 countries, and in total 1295 computer systems were compromised.
- **Operation Aurora:** This is a cyber attack, which began in mid-December 2009 and continued into February 2010. Google first publicly disclosed the attack on January 12, 2010, when Google stated that over 20 other companies had been attacked.
- **Stuxnet:** Stuxnet is a Windows-specific computer worm first discovered in June 2010. It is the first discovered worm that spies on and reprograms industrial systems.
- **Red October:** This cyber espionage network discovered in 2012 focused upon diplomatic, governmental and scientific research organizations in different countries mostly within the region of Eastern Europe, former USSR members and countries in central Asia.

The traditional approach to proxies says that they are placed at the main ingress and egress points on a network. Due to bandwidth requirements this approach fails to fully protect network services at the interaction level. Instead, it focuses upon the utilization of signatures and statistical analysis of data flows. What is required is a proxy approach targeted at the application layer within which a service functions and focused on the nature of the discourse that a server engages in with a client.

2. Universe of Discourse

We can analyze the search spaces required by various cyber attacks from a logical perspective with a view to identifying the complexity of the search spaces. So let the set of all possible inputs to a program be denoted as the set I_p. We will define two further sets: let I_a be the input set of all possible attacks, and let I_v be the input set of all possible valid inputs that are not an attack. Thus we may assert that $I_p = I_a \cup I_v$. As was observed in [1], the size and complexity of this search is to all intents and purposes infinity. The challenge for an intrusion detection system is to identify all malicious input from this set. Misuse Intrusion Detection Systems such assume that the input space is benign and that the set of valid attacks is finite, and thus $|I_a| < |I_v|$. If this statement is true then a logical deduction is that it is possible to express via a formal grammar and a set of rules by which the membership of I_v can be completely defined. In taking such a view, the implicit assumption is to allow all input except that which can be shown to be malicious. However, this assumption is now being challenged and event correlation models are being proposed that view the set of all possible attacks as being infinitely large and the set of all possible valid inputs being finite. Thus $|I_a| > |I_v|$. If this is true then we would assert that the logical approach to take to intrusion detection is to specify the set of allowable inputs and define all other non-valid input to be attacks, hence $I_v = I_p - I_a$. In taking such a view the implicit assumption is to deny all inputs except that which can be shown to be non-malicious. Consequently this project seeks to explore this novel approach to intrusion detection via the profiling of code artifacts to construct/derive a rule-set that defines the set I_v. In defining $|I_a| < |I_v|$ we are constraining the freedom of movement of our adversary and thus controlling the cyber battle space. Thus the challenge is to define the allowable inputs that can occur between a remote user and a service executing on a computer system. In defining this interaction a detailed model of the communication that can occur at the application (OSI level 7) must be constructed, hence the need to examine the service at the binary, and code, artifact level to determine the precise structure of the interaction.

3. State of the Art

3.1. Static Analysis of Assembly Code

Understanding what an executable does is paramount to the analysis of computer systems and networks in predicting accurately their behaviour, and to the discovery of critical vulnerabilities that have a devastating effect on our global computing infrastructure. Static approaches have also been applied to virus and worm detection, as well as polymorphic worm detection. Static analysis has also been applied to rootkit detection and to identifying spyware-like behaviour.

In addition we can also further classify these approaches based on the following [2]:

- Testing-based approaches try to trigger vulnerability by exercising an application with random or malicious inputs, [2].
- Monitoring-based approaches instead of examine the execution of an application during normal use look for anomalous behaviours. In particular, a whole area of research has focused on ways to detect attacks on the basis of the analysis of the system call invocations performed by a program [3].
- Slicing is a widely used technique for program analysis, debugging, and comprehension [4]. Software slice analysis attempts to approximate the state of an application during execution via the construction of cohesion and coupling metrics.
- Shape analysis attempts to construct a directed graph construct that represents the execution of a program [5]. Various topological transformation functions can be applied to identify sub-graphs, where each sub-graph corresponds to a known function call implementation.
- Alias analysis is a method for the identification of possible security flaws due to the conversion of one type or another, such as type conversation/casting in the C programming language.

Automated static analysis (ASA) identifies probable source code anomalies early in the soft ware development lifecycle that could lead to vulnerabilities. Automated static analysis (ASA) can identify common coding problems early in the development process via a tool that automates the inspection of source code [6]. Automated static analysis reports potential source code anomalies, which we call alerts, like null pointer dereferences, buffer over flows, and style inconsistencies [7]. Developers inspect each alert to determine if the alert is an indication of an anomaly important enough for the developer to fix. If a developer determines the alert is an important, fixable anomaly, then we call the alert an actionable alert [8]. When an alert is not an indication of an actual code anomaly or the alert is deemed unimportant to the developer (e.g. the alert indicates a source code anomaly inconsequential to the program's functionality as perceived by the developer), we call the alert an un actionable alert [8].

Improving ASA's ability to generate predominantly actionable alerts through the development of tools that are both sound and complete can be said to be an intractable problem [9]. Additionally, the development of algorithms underlying ASA requires a trade-off between the level of analysis and execution time [9]. Methods proposed for improving static analysis include annotations, which could be specified incorrectly and require developer overhead, and allowing the developer to select ASA properties, such as alert types, specific to their development environment and project [10]. Another way to increase the number of actionable alerts identified by static analysis is to use the alerts generated by ASA with other information about the software under analysis to prioritize or classify alerts.

Static analysis techniques attempt to identify the presence of a vulnerability, or the ways in which a vulnerability can be exploited. Thus in terms of search spaces they define $|I_A| < |I_V|$ to be true.

3.2. Dynamic Analysis

Dynamic analysis is the ability to observe and examine code as it executes in real-time, and thus it has become a central tool in computer security and vulnerability research. Dynamic analysis is attractive because it provides us with the ability to perform deductive reasoning about actual executions, and thus the ability to perform precise security analysis based upon run-time data.

Two of the most commonly employed dynamic analysis techniques in security research are dynamic taint analysis and forward symbolic execution. Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input. Symbolic execution refers to the analysis of programs by tracking symbolic rather than actual values, a case of abstract interpretation. Dynamic forward symbolic execution automatically builds a logical formula describing a program execution path, which reduces the problem of reasoning about the execution to the domain of logic. The two analyses can be used in conjunction to build formulas representing only the parts of an execution that depend upon tainted values [11].

The purpose of dynamic taint analysis is to track information flows [12]. Any program value whose computation depends on data derived from a taint source is considered tainted. Any other value is considered untainted.

Symbolic execution maintains a symbolic state, which maps variables to symbolic expressions. Forward symbolic execution allows us to reason about the behaviour of a program on many different inputs at one time by building a logical formula that represents a program execution. Thus, reasoning about the behaviour of the program can be reduced to the domain of logic. The advantages of forward symbolic execution include [13]:

- Multiple inputs. One of the advantages of forward symbolic execution is that it can be used to reason about more than one input at once.
- The primary difference between forward symbolic execution and regular execution is that when a function is executed it is evaluated symbolically; it returns a symbol instead of a concrete value. When a new symbol is first returned, there are no constraints on its value; it represents any possible value.
- Creating a forward symbolic execution engine is conceptually a very simple process: take the operational semantics of the language and change the definition of a value to include symbolic expressions.

The number of security applications utilizing these two techniques is enormous. Examples of security research areas employing either dynamic taint analysis of forward symbolic executions are:

- Unknown Vulnerability Detection. Dynamic taint analysis can look for misuses of user input during an execution. For example, dynamic taint analysis can be used to prevent code injection attacks by monitoring whether user input is executed [14].
- Automatic Input Filter Generation. Forward symbolic execution can be used to automatically generate input filters that detect and remove exploits from the input stream [15]. Filters generated in response to actual executions are attractive because they provide strong accuracy guarantees [17].
- Malware Analysis. Taint analysis and forward symbolic execution are used to analyse how information flows through a malware binary, explore trigger-based behaviour, and detect emulators [16].
- Test Case Generation. Taint analysis and forward symbolic execution are used to automatically generate inputs to test programs [17], and can generate inputs that cause two implementations of the same protocol to behave differently [17].
- Abstract Interpretation [18] using numerical domains, such as intervals, octagons, and convex polyhedrals necessitates the use of widening and narrowing operators to guarantee termination over loops in the program.

It should be noted that taint analysis as a technique for malware detection suffers from a variety of evasion techniques that allow an intruder to avoid detection of the malicious software. Dynamic analysis techniques attempt to identify the presence of a vulnerability, or the ways in which a vulnerability can be exploited. Thus in terms of search spaces they define $|I_A| < |I_V|$ to be true.

3.3. Intrusion Detection

In terms of the analysis of software artifacts various Intrusion Detection Systems have been developed that attempt to detect the presence of software artifacts, such as malicious software, at the TCP/IP packet level. Examples of such approaches include:

- The Snort intrusion detection system is a host/network rule based expert system approach to misuse detection. It operates by monitoring all TCP/IP traffic targeted at a host/network. Consequently, the rule based expert system approach to misuse detection is incapable of detecting new forms of malicious software due to its lack of prior knowledge.
- Process monitoring approaches such as [19] monitor software processes with a view to identifying malicious activity based on pre-defined norms of behaviour. This type of approach is based on user-defined levels of normality and has a computational impact on the performance of the process being monitored. Anomaly detection approaches such as [20], attempt to learn user behaviour via Markov Chains based analysis engines. However such approaches suffer from generating false positive results and the time taken to learn behaviour.
- A proxy functions as the man-in-the-middle of an information flow, allowing for the information stream to be inspected and rules governing malicious behaviour applied [21]. Such approaches either make use of user defined rules to identify malicious activity, or else make use of learning algorithms to identify abnormal behaviour. In addition, proxies implement a white-list/black-list access control mechanism governing access to remote systems. Due to the volume of network traffic that the network-based proxy solutions support security checks governing malicious behaviour are computationally and time/resource bounded. The net effect of this is that the complexity and sophistication of the rule set/language is constrained. Proxies' function to protect systems from attacks at the application/content layer. The challenges facing this type of approach is that a rule specification is required to define malicious behaviour, and that learning algorithms suffer from generating false positive results and the time taken to learn behaviour. As a rule application/content layer proxies implement a default allow rule. Thus proxies fail to detect novel sophisticated attacks.

All of the above attempt to define rules/algorithms by which erroneous behaviour can be identified and thus in terms of search spaces they define $|I_A| < |I_V|$ to be true. Therefore they attempt to codify all of the ways in which a threat agent can exploit vulnerability in order to gain unauthorized access to a system. The aim of the proposed architecture is to limit the access space within which an adversary can manoeuvre by assuming that all input is malicious unless it meets a goodness function. Hence in terms of search spaces they define $|I_A| < |I_V|$ to be false and $|I_A| > |I_V|$ to be true. Thus the default access rule is to deny access from a remote user to a server/service.

4. Conceptal Architecture

The basic principle underpinning the micro proxy conceptual architecture is to assume that all input are malicious unless proved otherwise, in effect to implement a default deny policy at the application level. In making such an assumption we are constraining the way in which an adversary can interact with a system. For any interaction to succeed it must pass a series of tests that define the goodness of the interaction. Hence in terms of search spaces we define $|I_A| > |I_V|$. The challenge can thus be reshaped to the analysis of software artifacts so as to define/impose the rules governing the acceptance of valid input.

Figure 1 Defines the Conceptual Architecture. The role and function of **Figure 1** is to define the various components by which a binary/software artifact is analyzed and a proxy configuration is constructed. The basic ontological constructs contained in **Figure 1** are that arrows represent data flows and, squares represent processes, ovals represent remote agents (either a user or a service) and rounded squares represent data sources. At the heart of the conceptual architecture is a control flow graph.

4.1. The Service Analysis Engine

The service analysis engine takes a set of Service Area Artifacts as input and produces a service area profile. For example, suppose that the remote/legacy service is a web server and that the files in the home directory of the web server are as follows:

1) /index.html
2) /search.php
3) /images/logo.jpeg

The service area artifacts listed above function to specify the allowable set of valid file requests that can be contained in a HTTP request.

4.2. The Reverse Analysis Engine

The role and function of the reverse analysis engine is to take either binary artifacts such as PE/ELF, or source code artifacts such as C, and to produce a control flow graph. A control flow graph is a directed graph that represents all of the possible execution paths that can occur during the execution of a program [22]. Certain compliers such as GCC and disassemblers already possess a limited capability to construct control flow graphs in a simple text based format [23].

4.3. The Control Flow Graph (CFG)

The role and function of a control flow graph is to allow for the easy constructing and reasoning about the control flow of an artifact. Traditional control flow models are no more than type-less directed graphs where each node is either a collection of commands or a conditional statement. The control flow model presented, and utilized, in this project draws upon socio-linguistic constructs to create a richer and more structured syntactic/semantic representation of the execution of an artifact. **Figure 2** is a simple example of a C program and a control flow graph that determines if the character typed at the keyboard is a space or not. A control flow graph is a directed graph that consists of a number of nodes and arcs. There are a number of rules governing the consistency of a control flow graph.

- Every arc in a control flow graph must have a conditional associated with it.
- Every node is either an atomic action or a function that can be decomposed. In **Figure 2**, we can see one atomic action that is assigned the result value of a function to a variable, and three functional actions.
- The logical conjugation of all of the conditions on the output arcs from a node is logically true. So for example in **Figure 2** we can observe that the following logical condition is true: $a = 31 \land a \neq 32$.
- Every arc must have a source and a destination. The source of an arc is called the point from which the arc

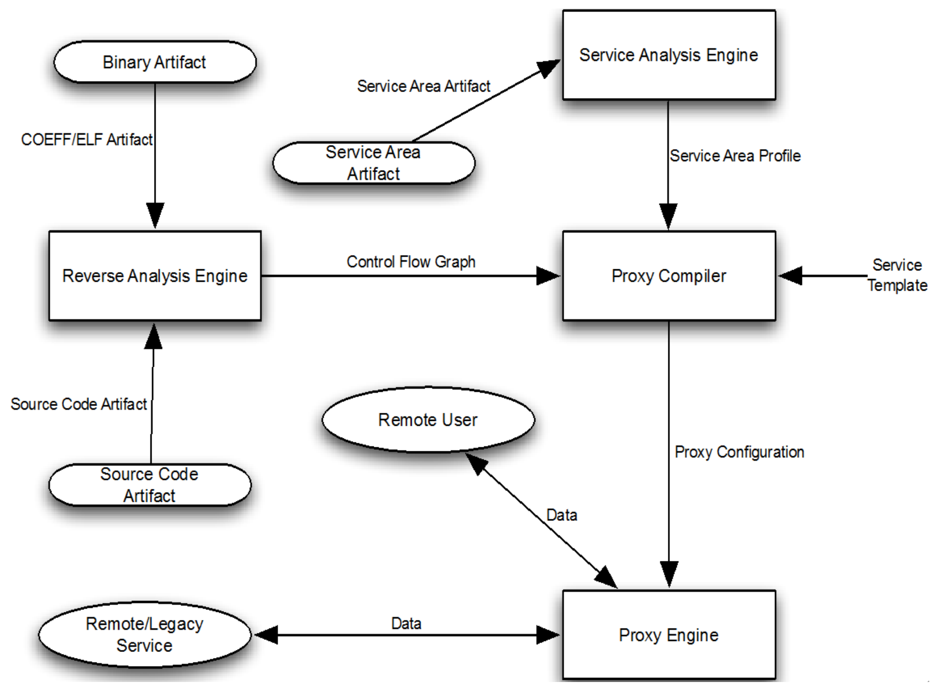
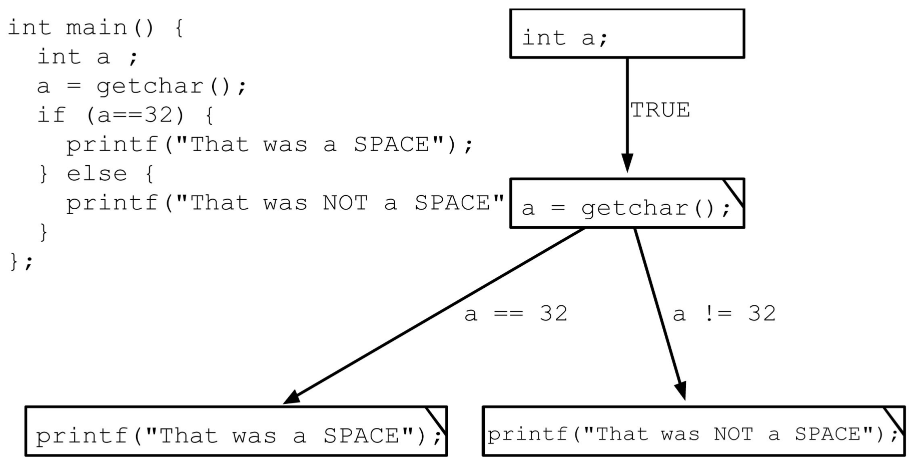


Figure 1. Conceptual architecture.



```
int main() {
  int a ;
  a = getchar();
  if (a==32) {
    printf("That was a SPACE");
  } else {
    printf("That was NOT a SPACE"
  }
};
```

Figure 2. A simple control flow graph.

originates and the destination of an arc is the point at which the arc terminates.

- Only one node in the graph can function solely as the source for arcs and that node is called the root node and is the logical start of the control flow graph.

In constructing a socio-linguistic model of a control flow graph it is necessary to map a computer programming language construct into a series of speech acts. A speech act is an action that when combined within a series of rules allows for the expression of a conversation. Thus when viewing the following program/server we may observe that it presents a conversation in content and structure [24]. In expressing this content and structure in accordance with the formal definition of a speech act we map certain types of structures to program constructs.

- A propositional act is a statement which can be evaluated to be true or false, such as “is x + y < z” or “does this file exist”. People or machines can meaningfully utter propositional acts. Thus in terms of a control flow graph propositional acts are viewed as directed arcs that link nodes together.
- An illocutionary act is always performed when a person utters certain expressions with an *intention*. Thus we

can view a function invocation as an illocutionary act that can be seen as an expression of intent to perform some behaviour. Types of illocutionary act are:

- o Assertive: speech acts that commit a speaker to the truth of the expressed such as assigning a value to a variable (x = x + y).
- o Directive: speech acts that are to cause the agent to take a particular action, e.g. the expression of a conditional statement such as an if, case, repeat or while statement.
- o Commissive: speech acts that commit an agent to some future action, e.g. a function innovation (max(a,b)).
- o Declaration: speech acts that change the reality in accordance with the proposition of the declaration, e.g. the termination of a loop, or the return value of a function.
- o Expressive: speech acts that express the agent's positions towards the proposition. In programming terms it can be viewed as the creation of a variable, e.g. int a.

- A perlocutionary act is an act that produces an effect on the behaviour of an agent. For example in control flow graph terms a perlocutionary act can be seen as an action such as input or output.

The output of the reverse analysis engine is a control flow graph such as the one described in **Figure 2**, but encoded in XML.

4.4. The Proxy Compiler

The proxy compiler functions to take the following as input and to produce an executable configuration as output.

- Service Area Profile (SAP)
- Control Flow Graph (CFG)

The executable configuration is produced via the concurrent execution of a series of defined constraints on a distributed cloud environment. This configuration is then executed on the proxy engine. The proxy compiler checks that the structure provided by the control flow graph matches the parameter provided via the analysis of the file/directory structures (the service area profile), and the application interaction patterns/behaviours specified by the service template.

4.5. The Proxy Engine

The role of the proxy engine is to function as a guard ensuring that all data passing to/from the remote service is protected and meets the goodness criteria as defined in the proxy configuration. Thus the proxy takes the input data stream from the user of the service and executes the configuration of proxy to ensure that the input data stream adheres to the rules governing the goodness of the input. If the goodness conditions are adhered to then the data is passed onto the remote service.

5. Evaluation

The test methodology to evaluate the utility of the proxy architecture is to take all of the Microsoft Security Bulletins published from 1st Jan 2011 to 31st Dec 2011 and examine via inspection of each bulletin whether the proxy architecture could detect the attack. The role and function of the Microsoft Security Bulletins is to provide an independent data set from which an impartial and repeatable assessment of the utility of the proxy architecture could be conducted. From Jan 2011 to Dec 2011 Microsoft published 100 security bulletins numbered MS11-001 to MS11-100. These security bulletins cover every type of Microsoft technology, operating system and application. In board terms we can classify these security bulletins via two dimensions:

- How the exploit code is delivered. Typically there are two ways that exploit code is delivered to a target either via a remote connection of some description or locally.
- The target of the exploitation. Typically this can be divided down into four elements.
    - o Applications such as IE, Word, PowerPoint, etc.
    - o Middle-ware such as .NET, JavaScript, OLE etc.
    - o Services such as HTTP, DNS, WINS etc.
    - o Kernel Executive, Host Operating System, Device Drivers, TCP/IP Stack etc.

The test data contained in the Microsoft Security Bulletins is broken down as follows (**Table 1**).

Table 1. Evaluation data.

	Application	Middleware	Services	Kernel
Remote exploitation	32	24	19	7
Local exploitation	0	0	0	18

Thus from the data contained in **Table 1** we can observe for all of 2011 Microsoft issues:

- 18 security bulletins regarding kernel level vulnerabilities that could be exploited locally and 7 security bulletins regarding kernel vulnerabilities that could be exploited remotely;
- 19 security bulletins regarding service level vulnerabilities that could be exploited remotely;
- 24 security bulletins regarding middleware level vulnerabilities that could be exploited remotely;
- 32 security bulletins regarding application level vulnerabilities that could be exploited remotely.

From the analysis of the data we can observe for all Microsoft Security Bulletins published in 2011 (MS 001 to MS 100), that the proxy could have detected 29 of these vulnerabilities, and have their exploitation prevented by our proxy architecture. Of these 29 vulnerabilities that could have been detected detailed breakdown is as follows:

- 19 of these vulnerabilities relate to remote exploitation of services;
- 2 of the vulnerabilities relate to remote exploitation of the kernel;
- 8 of these vulnerabilities relate to remote exploitation of the middleware.

It should be noted that our proposed micro-proxy architecture;

- Did detect 100% of the vulnerabilities targeted at a service running on a server. Key services include: DNS, WINS, IIS and FTP.
- Could only detect the middleware vulnerabilities when the middle-ware technology was deployed within a server environment;
- Did not detect any of the Microsoft Security Bulletins relating to applications such as Microsoft Office and Internet Explorer.

Acknowledgements

My thanks go to the following people for their help: Dr. Gaius Mulley and Mr. Konstantinos Xynos. This research has been funded by DSTL.

References

[1] Bass, T. (2000) Intrusion Detection Systems and Multi-Sensor Data Fusion. *Communications of the ACM*, **43**, 4. http://dx.doi.org/10.1145/332051.332079

[2] Cova, M., Felmetsger, V., Banks, G. and Vigna, G. (2006) Static Detection of Vulnerabilities in x86 Executables. *ACSAC* '06 *Proceedings of the* 22nd *Annual Computer Security Applications Conference*, Miami Beach, December 2006, 269-278.

[3] Mutz, D., Valeur, F., Vigna, G. and Kruegel, C. (2006) Anomalous System Call Detection. *ACM Transactions on Information and System Security*, **9**, 1. http://dx.doi.org/10.1145/1127345.1127348

[4] Wang, T. and Roychoudhury, A. (2007) Hierarchical Dynamic Slicing. *Proceedings of the* 2007 *International Symposium on Software Testing and Analysis*, 228-238.

[5] Zhang, X. and Gupta, R. (2004) Whole Execution Traces. 37*th International Symposium on Microarchitectures*. IEEE Press.

[6] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. and Vouk, M. (2006) On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*, **32**, 240-253. http://dx.doi.org/10.1109/TSE.2006.38

[7] Hovemeyer, D. and Pugh, W. (2004) Finding Bugs Is Easy. *Proceedings of the* 19*th ACM Conference on Object-Oriented Programming*, *Systems*, *Languages*, *and Applications*, Vancouver.

[8] Heckman, S. and Williams, L. (2009) A Model Building Process for Identifying Actionable Static Analysis Alerts. *Proceedings of the* 2nd *IEEE International Conference on Software Testing*, *Verification and Validation*, Denver, 1-4 April 2009, 161-170.

[9] Chess, B. and McGraw, G. (2004) Static Analysis for Security. *IEEE Security & Privacy*, **2**, 76-79. http://dx.doi.org/10.1109/MSP.2004.111

[10] Yi, K., Choi, H., Kim, J. and Kim, Y. (2007) An Empirical Study on Classification Methods for Alarms from a Bug-Finding Static C Analyzer. *Information Processing Letters*, **102**, 118-123. http://dx.doi.org/10.1016/j.ipl.2006.11.004

[11] Schwartz, E.J., Avgerinos, T. and Brumley, D. (2010) All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). *Proceedings of the* 2010 *IEEE Symposium on Security and Privacy*, Oakland, 16-19 May 2010, 317-331. http://dx.doi.org/10.1109/SP.2010.26

[12] Cavallaro, L., Saxena, P. and Sekar, R. (2008) On the Limits of Information Flow Techniques for Malware Analysis and Containment. *Proceedings of the* 5*th International Conference on Detection of Intrusions and Malware*, *and Vulnerability Assessment* (*DIMVA*), Springer.

[13] Pasareanu, C.S. and Visser, W. (2009) A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal of Tools Technology Transfer*, **11**, 339-353. http://dx.doi.org/10.1007/s10009-009-0118-1

[14] Crandall, J., Su, Z., Wu, S.F. and Chong, F. (2005) On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *Proceedings of the ACM Conference on Computer and Communications Security*, 235-248. http://dx.doi.org/10.1145/1102120.1102152

[15] Brumley, D., Newsome, J., Song, D., Wang, H. and Jha, S. (2008) Theory and Techniques for Automatic Generation of Vulnerability Based Signatures. *IEEE Transactions on Dependable and Secure Computing*, **5**, 224-241. http://dx.doi.org/10.1109/TDSC.2008.55

[16] Sharif, M., Lanzi, A., Giffin, J. and Lee, W. (2009) Automatic Reverse Engineering of Malware Emulators. *Proceedings of the IEEE Symposium on Security and Privacy*, 94-109.

[17] Cadar, C., Dunbar, D. and Engler, D. (2008) Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 209-224.

[18] Mine, A. (2001) A New Numerical Abstract Domain Based on Difference-Bound Matrices. *PADO II*, **2053**, 155-172.

[19] Yin, H., Poosankam, P., Hanna, S. and Song, D. (2010) HookScout: Proactive Binary-Centric Hook Detection. *Proceedings of the Detection of Intrusions and Malware*, *and Vulnerability Assessment*, Springer, 1-20.

[20] Frossi, A., Maggi, F., Rizzo, G.L. and Zaneo, S. (2009) Selecting and Improving System Call Models for Anomaly Detection. *Proceedings of the Detection of Intrusions and Malware*, *and Vulnerability Assessment*, Springer.

[21] Bockermann, C., Apel, M. and Meier, M. (2009) Learning SQL for Database Intrusion Detection Using Context-Sensitive Modelling. *Proceedings of the Detection of Intrusions and Malware*, *and Vulnerability Assessment*, Springer.

[22] Tanenbaum, A.S. (2007) Modern Operating Systems, Pearson Education.

[23] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J. (2009) Control Flow Integrity Principles, Implementations and Applications. *ACM Transactions on Information and Systems Security* (*TISSEC*), **13**, 1. http://dx.doi.org/10.1145/1609956.1609960

[24] Searle, J.R. (1969) Speech Acts: An Essay in the Philosophy of Language. Cambridge Press, Cambridge. http://dx.doi.org/10.1017/CBO9781139173438