SANDIA REPORT

SAND2011-7055

Unlimited Release

Printed September 2011

Indirection and Computer Security

Michael J. Berg

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America.

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

Indirection and Computer Security

Michael J. Berg

Networked Systems Survivability and Assurance
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0672

Abstract

The discipline of computer science is built on indirection. David Wheeler famously said, “All problems in computer science can be solved by another layer of indirection. But that usually will create another problem” [1].

We propose that every computer security vulnerability is yet another problem created by the indirections in system designs and that focusing on the indirections involved is a better way to design, evaluate, and compare security solutions. We are not proposing that indirection be avoided when solving problems, but that understanding the relationships between indirections and vulnerabilities is key to securing computer systems.

Using this perspective, we analyze common vulnerabilities that plague our computer systems, consider the effectiveness of currently available security solutions, and propose several new security solutions.

Contents

1	Foundations	7
1.1	What is a Computer?	7
1.1.1	Abstract Machine Hierarchies	7
1.2	What is Security?	8
1.2.1	Policies, Policy Violations, and Vulnerabilities	8
1.2.2	Security and Insecurity	9
2	Indirection and Insecurity	10
2.1	Mapping	11
2.2	Abstraction	11
3	Case Studies	12
3.1	Instructions versus Data	12
3.1.1	Converting Data into Instructions	13
3.1.2	Applications versus Data Files	13
3.1.3	Processor Instructions versus Data	14
3.1.4	SQL Instructions versus Data	15
3.1.5	JavaScript Instructions versus HTML Data	16
3.2	Pointers	17
3.2.1	Remove Pointer Access	17
3.2.2	Pointer Value Checks	17
3.2.3	Address Space Layout Randomization	17
3.2.4	Pointer Encryption	17
3.3	Data Structures versus Unstructured Memory	18
3.3.1	Remove Unstructured Memory Access	18
3.3.2	Stack Smashing Protection	18
3.4	Integer Errors	21
3.4.1	Multiple-Precision Integers	22
3.4.2	Run-Time Error Detection	22
3.5	The Network Stack	22
3.5.1	CAM Table Flooding	23
3.5.2	ARP and NDP Redirection	24
3.5.3	DNS Redirection	25
3.5.4	Security Solutions that Cross Layers of Indirection	25
3.6	The Hardware and Software Stack	25
3.6.1	Operating System Rootkits	26
3.6.2	Malicious Virtual Machine Managers	26
3.6.3	Malicious Firmware	26
3.6.4	Malicious Hardware	26
3.6.5	Security Solutions that Cross Layers of Indirection	27
4	Conclusion	28

List of Examples

3.1	Vulnerable HTML document.	16
3.2	Vulnerable C program.	19

List of Figures

1.1	Abstract machine hierarchy.	8
3.1	i386 stack: Unprotected.	19
3.2	i386 stack: ProPolice.	20
3.3	OSI reference model.	23
3.4	Hardware/Software stack.	26

List of Tables

3.1	Integer operations.	21
-----	--------------------------	----

1 Foundations

The topic of this report is computer security. We must start by answering two questions:

- What is a computer?
- What is security?

These may seem like trivial questions to which everyone has answers. However, everyone seems to have different answers for these questions and different definitions of the terms involved in describing them. Any field of science needs a common set of terms to enable the unambiguous communication of ideas. This section defines computer security terms based on fundamental principles of computer science, set theory, and definitions developed by other researchers.

1.1 What is a Computer?

A computer is any system that is computationally equivalent to a bounded-tape universal Turing machine¹. Desktop computers, laptop computers, servers, virtual machines, video game consoles, personal digital assistants (PDA), cell phones, and the microcontrollers in a multitude of products ranging from toaster ovens to automobiles are all computers by this definition. The methods presented in this report for reasoning about vulnerabilities and effective solutions are applicable to any of these computers.

1.1.1 Abstract Machine Hierarchies

We will represent any machine that is equivalent to a universal Turing machine as an abstract machine M .

Abstract machines $M_0, M_1, \dots, M_n$ and abstract program sets $P_0, P_1, \dots, P_n$ are defined in [3] such that, for $0 \leq i \leq n$, P_i is a program set that runs on machine M_i . The combination of abstract machine M_i with abstract program set P_i , written as $\langle M_i, P_i \rangle$, implements the abstract machine M_{i+1} , where $0 \leq i < n$. Each program set P_i implements a mapping function between the state spaces of M_i and M_{i+1} . The abstract machine hierarchy terminates at M_n because program set P_n either does not exist or does not define a new Turing-equivalent machine.

For example, let the hardware platform define M_0 and let the firmware and operating system kernel be in program set P_0 . The combination $\langle M_0, P_0 \rangle$ defines M_1 , which is commonly known as user-space. Interpreters or run-time environments for languages such as Perl, Python, Ruby, Visual Basic, or Java each define a program set P_1 that runs on M_1 . Each combination $\langle M_1, P_1 \rangle$ defines a new M_2 , the abstract machine accepting that interpreted language or byte-code. Similarly, any virtual machine manager that runs on M_1 is in a program set $\hat{P}_1$. The combination $\langle M_1, \hat{P}_1 \rangle$ defines $\hat{M}_2$, a virtual machine. An operating system $\hat{P}_2$ can then run on the virtual machine $\hat{M}_2$, defining a new user-space $\hat{M}_3$, and so on. The example abstract machine hierarchy just described is depicted in Figure 1.1.

¹ Any reader unfamiliar with Turing machines should consult a textbook on automata theory, such as [2].

Figure 1.1 Abstract machine hierarchy.

Understanding this hierarchy of abstract machines is critical because similar vulnerabilities occur throughout the hierarchy. Vulnerabilities in different abstract machines M_i and M_j may currently be viewed as different classes of vulnerabilities, while they are actually instances of the same vulnerability at different locations in the hierarchy of abstract machines. Principles used to mitigate vulnerabilities in M_i may work to mitigate vulnerabilities in M_j . Section 3 includes demonstrations of how existing defenses can be translated to new locations in the abstract machine hierarchy.

1.2 What is Security?

Formal definitions of security concepts provide the foundation for reasoning about what a vulnerability is, what security is, and how effective any given security solution can be.

1.2.1 Policies, Policy Violations, and Vulnerabilities

Informally, a policy defines what is allowed and disallowed for a system; a policy violation occurs when something was allowed that should have been disallowed or when something was disallowed that should have been allowed; and a vulnerability is the set of conditions that allowed the policy violation. Formal definitions for security policies, policy violations, and vulnerabilities are presented in [4]. These definitions are summarized here for their usefulness in defining security and insecurity.

Policy The function $\mathbf{P} : S \times O \times A \rightarrow \{\text{allowed}, \text{disallowed}\}$ such that for any subject $s \in S$, object $o \in O$, and action $a \in A$, $\mathbf{P}(s, o, a) = \text{allowed}$ if a on o by s is allowed, otherwise $\mathbf{P}(s, o, a) = \text{disallowed}$. [4]

Additionally, the following four levels of policy are defined:

An *Oracle Policy* (OP) is the theoretical ideal that encapsulates perfect knowledge of the system and the system owner's intent. The OP is what the system owner wants the system to do and is not tied to any specific implementation [4].

A *Feasible Oracle Policy* (FP) is an approximation of the OP that is feasible to implement on system x due to the resource limitations of system x or the general limitations of what is computable [4].

The *Configured Oracle Policy* (CP) is the implementation on system x of the FP. The CP is what system x is configured to do [4].

The *Actual Policy* (AP) is what the implementation of system x actually allows to happen [4].

Policy Violation Any tuple $\langle s, o, a \rangle$ where $s \in S$, $o \in O$, $a \in A$ such that $\mathbf{OP}(s, o, a) \neq \mathbf{FP}_x(s, o, a)$, $\mathbf{FP}_x(s, o, a) \neq \mathbf{CP}_x(s, o, a)$, or $\mathbf{CP}_x(s, o, a) \neq \mathbf{AP}_x(s, o, a)$. [4]

Vulnerability Any tuple $\langle C_x, V_x \rangle$ where C_x is the nonempty set of conditions that enables the nonempty set of policy violations V_x for some system x . [4]

1.2.2 Security and Insecurity

A large portion of the computer security community tries to use absolute definitions of security and insecurity such as:

Secure A system is secure if that system has no vulnerabilities.

Insecure A system is insecure if that system has one or more vulnerabilities.

While it is desirable to have a system that is proven secure or to have absolute security metrics for comparing different systems, both of these desires have been proven to be unobtainable [5] [6].

An axiomatic approach is taken in [5] that considers sets of known and unknown vulnerabilities for system owners and all possible adversaries, sets of mitigated and unmitigated vulnerabilities, and the intersections of these sets. The result of the axiomatic reasoning is that absolute measures of security are not possible. However, [5] demonstrates that while a system can not be proven to be secure, if no adversary knows any unmitigated vulnerabilities, a system may be secure in practice.

Secure in Practice A system is secure in practice if that system has no vulnerabilities that are known to and exploitable by any adversary [5].

Using an approach based on automata, [6] defines two conceptual languages: *SECURE* and *INSECURE*². A series of proofs based on Turing machines and computability are then outlined with two primary results. First, proving that a system is secure is an undecidable problem. Second, determining if a system is currently in a secure state is a decidable problem. The real-time determination of whether a system is currently in a secure state enabled a useful definition of security.

Real-time Secure A system is real-time secure if and only if every configuration in its computation history is authorized by the security policy [6].

The results from both [5] and [6] are that systems can not be proven to be secure. A system can be proven to be insecure by demonstrating a vulnerability, but the strongest security statement possible is that a system *is not known to be insecure*.

Since the ideal of a provably secure system is not possible, the security community must focus on practical issues of designing and building systems that are secure in practice. In support of designing systems that are secure in practice, there are various questions that can be considered: What are the known ways for systems to be insecure? How can we eliminate, mitigate, or at least constrain entire classes of known insecurities? What real-time security checks can be effectively implemented? These questions and a new way of looking at these questions are the focus of the remainder of this report.

² The language *INSECURE* is defined as the complement of the language *SECURE* [6].

2 Indirection and Insecurity

Indirection Indirect action or procedure. Not directly achieved [7].

The discipline of computer science is built on indirection. David Wheeler famously said, “All problems in computer science can be solved by another layer of indirection. But that usually will create another problem” [1]. The problems created by indirection are normally described as execution overhead (in time or space) and design complexity. Consider the abstract machine hierarchy presented in Section 1.1.1. Each program set P_i is a layer of indirection that enables program set P_n to indirectly run on the physical machine M_0 . However, each P_i requires storage space and execution time on abstract machine M_i , which translates into storage space and execution time on the physical machine M_0 .

We propose that all computer security vulnerabilities are problems created by indirection. Indirections enable or even provide the conditions that are required for vulnerabilities³. We are not proposing that indirection be avoided when solving problems, but that understanding the relationships between indirections and vulnerabilities is key to securing computer systems. The use of indirection does not necessitate the existence of vulnerabilities; however, Section 3 analyzes a variety of indirections that have a propensity to produce vulnerabilities.

A better way to design security solutions is to identify the indirections involved and implement methods for eliminating or mitigating the mismatch between the indirection and reality. In decreasing order of desirability, the spectrum of options are

1. Make the indirection and reality match by changing the indirection to match reality, changing reality to match the indirection, or changing both indirection and reality to match. Making the indirection and reality match is an *inherent security solution* since the root cause of the vulnerability has been removed. Inherent security solutions eliminate possibilities for $\mathbf{OP}(s, o, a) \neq \mathbf{FP}_x(s, o, a)$, $\mathbf{FP}_x(s, o, a) \neq \mathbf{CP}_x(s, o, a)$, or $\mathbf{CP}_x(s, o, a) \neq \mathbf{AP}_x(s, o, a)$.
2. Add programmatic checks to ensure that the system is operating in the valid region of the indirection. If the system leaves the valid region of the indirection, a fatal error has occurred and the system is stopped. This method is a programmatic approximation of an inherent security solution and will be referred to as a *programmatic security solution*. The root cause of the vulnerability is still present, but programmatic checks are in place to detect and prevent attempts to exploit the vulnerability. Programmatic security solutions contribute to the real-time security of the system by increasing the number of configurations in the computation history that are checked against the security policy.
3. Make exploiting a vulnerable indirection computationally complex. This method is a probabilistic approximation of a programmatic security solution⁴ and will be referred to as a *probabilistic security solution* since the vulnerability can still be successfully exploited with some (hopefully very small) probability.

Many security solutions (such as SELinux, chroot jails, and segmented networks) attempt to contain or limit what can be done with the successful exploitation of a vulnerability. However, containing or limiting a successful exploit is accomplished by using at least one of the three methods above at a lower layer in

³ Examples are provided in Section 3.

⁴ The equivalence of diversity-based defenses to probabilistic, run-time type checking is shown in [8].

the abstract machine hierarchy. Consider again the abstract machine hierarchy presented in Section 1.1.1. Changing abstract machine M_i to include any of the above three methods changes the operating environment for program set P_i and, hence, abstract machine M_{i+1} . The program set P_i can still implement any M_{i+1} , but the external behavior of P_i is contained within the limits established by M_i .

There is often value in combining a programmatic security solution for a given vulnerability with a probabilistic security solution for the same vulnerability. The underlying vulnerability is still present with either of these solution types, so combining a programmatic security solution with a probabilistic security solution provides defense in depth against the vulnerability being exploited.

It is useful to consider two ways of describing indirection: mapping and abstraction.

2.1 Mapping

Mapping To assign (as a set or element) in a mathematical or exact correspondence. To be assigned in a relation or connection [7].

Mapping is establishing a connection between sets of things. In mathematics, a function defines a mapping from the input set to the output set and can be one-to-one, one-to-many, many-to-one, or many-to-many. Mappings do not require the input and output sets to be similar.

How a mapping is established is important. Static mappings are defined by the system designer or owner and remain constant. Dynamic mappings are established and periodically changed at run-time. Both static and dynamic mappings are used by adversaries. Adversaries influence the establishing of dynamic mappings to modify the operation of the system and rely on known static mappings to gain a foothold on a system. Some of the ways in which adversaries leverage both dynamic and static mappings are covered in Section 3.

2.2 Abstraction

Abstraction Disassociation from any specific instance. The act of considering apart from application to or association with a particular instance [7].

Abstraction is the generalizing or simplifying of a topic by removing details. Abstraction is what allows the hierarchy of processor registers, L1 cache, L2 cache, system RAM, and swap space to be treated as a single memory space by the programmer. Abstraction is what allows local file systems and networked file systems to appear the same to the system user. Abstractions emphasize how a set of things are similar to each other.

However, abstractions are not without defect. “All non-trivial abstractions, to some degree, are leaky” [9]; where *leaky* means that artifacts or side-effects of the implementation are still observable through the abstraction. For example, there are speed differences between registers, cache, RAM, and disk that are evident in how data and instruction locality impact performance. The differences in reliability and speed between local storage media and network traffic leak through the local versus networked file system abstraction. Whenever an abstraction is leaky, an adversary has the opportunity to obtain information about the system or to modify the operation of the system. Several leaky abstractions are considered in Section 3.

3 Case Studies

This section explores a variety of common indirections that have a history of producing vulnerabilities. Each case study focuses on a particular indirection, known vulnerabilities enabled by that indirection, and defenses against those vulnerabilities. The case studies are not exhaustive, but are intended to demonstrate how a wide variety of vulnerabilities and security solutions can be studied through the lens of indirection. Several new security solutions are also proposed to demonstrate how security solutions from one location in the abstract machine hierarchy can be translated to other locations in the abstract machine hierarchy.

3.1 Instructions versus Data

Instruction A code that tells a computer to perform a particular operation [7].

Data Information in numerical form that can be digitally transmitted or processed [7].

Programmers find it useful to make a distinction between instructions and data that is not inherently present in Turing machines, but that can be built into each Turing machine. Consider a Turing machine M that enforces a distinction between instructions and data by partitioning the set of states Q and the set of input symbols Σ as follows:

$$\begin{aligned} & \{ Q_{instructions} \subset Q, Q_{data} \subset Q \mid Q_{instructions} \cap Q_{data} = \emptyset \} \\ & \{ \Sigma_{instructions} \subset \Sigma, \Sigma_{data} \subset \Sigma \mid \Sigma_{instructions} \cap \Sigma_{data} = \emptyset \} \end{aligned} \quad (3.1)$$

In other words, states and tape symbols can be associated with either instructions or data, but not both.

The transition function δ includes transitions from states in $Q_{instructions}$ to states in Q_{data} and vice versa. For the current state q and the current input symbol σ , if $q \in Q_{instructions}$ and $\sigma \in \Sigma_{data}$, the Turing machine will reject the language for receiving a data symbol in an instruction accepting state. Similarly, if $q \in Q_{data}$ and $\sigma \in \Sigma_{instructions}$, the Turing machine will reject the language for receiving an instruction symbol in a data accepting state. In this way, M_i can enforce the distinction between instructions and data on any program P_i running on M_i . However, M_i can not, in the general case, enforce a distinction between instructions and data beyond P_i to M_{i+1} .

Proof: Let M_i be a universal Turing machine that distinguishes between instructions and data by partitioning Q and Σ as shown in Equation (3.1). The machine M_i , therefore, enforces a distinction between instructions and data on program set P_i . Let $\langle M_i, P_i \rangle$ implement a new Turing machine M_{i+1} . The program set P_i can encode the set of input symbols Σ for M_{i+1} using only symbols from Σ_{data} for M_i . Unless P_i enforces a new distinction between instructions and data, there will be no distinction between instructions and data in M_{i+1} . The distinction between instructions and data in M_i does not guarantee a distinction between instructions and data in M_{i+1} . $\square$

An alternate method of distinguishing between instructions and data is to have separate instruction and data tapes. However, the use of separate tapes does not change the conclusion that M_i can not guarantee a distinction between instructions and data in M_{i+1} .

Proof: Let M_i be a two-tape universal Turing machine where the first tape contains only instructions, the second tape contains only data, and instructions are only executed from the instruction tape. The machine M_i ,

therefore, enforces a distinction between instructions and data on program set P_i . Let $\langle M_i, P_i \rangle$ implement a new single-tape Turing machine M_{i+1} where the tape of M_{i+1} maps into the data tape of M_i . Both the instructions and data of the programs running on M_{i+1} will be stored on the data tape of M_i . Unless P_i enforces a new distinction between instructions and data, there will be no distinction between instructions and data in M_{i+1} . $\square$

Conversely, a Turing machine can be designed to enforce a distinction between instructions and data for the programs running on that Turing machine, even when the underlying universal Turing machine does not enforce a distinction between instructions and data.

Proof: Let M_i be a universal Turing machine that makes no distinction between instructions and data. Let $\langle M_i, P_i \rangle$ implement a new Turing machine M_{i+1} . The program set P_i can enable a distinction between instructions and data by defining separate instruction and data tapes for M_{i+1} or by partitioning the sets of states and input symbols for M_{i+1} as shown in Equation (3.1). The machine M_{i+1} enforces a distinction between instructions and data on program set P_{i+1} even though M_i makes no distinction between instructions and data. $\square$

Each Turing machine is responsible for enforcing its own separation between instructions and data.

The lack of distinction between instructions and data is one of the conditions that enables various security vulnerabilities. Classic buffer overflows place data in memory that are later treated as instructions by the processor. SQL injection presents data to the database engine (a machine that accepts SQL) that are later treated as SQL commands (i.e., instructions). Cross-site scripting places data on a web page that are later treated as JavaScript instructions by the user's web browser (a machine that accepts JavaScript). Processor-level code injection, SQL injection, and cross-site scripting are all instances of the same vulnerability in different layers of the abstract machine hierarchy.

3.1.1 Converting Data into Instructions

When instructions in M_{i+1} are encoded using data in M_i , P_i is the point of conversion from data to instructions. Examples include linkers that convert binary files into running processes and scripting language interpreters that convert text files into running processes. Frequently, the point of conversion in P_i is exposed as an interface in M_{i+1} , allowing programs in P_{i+1} to submit data to M_{i+1} for conversion to new instructions in P_{i+1} , potentially circumventing any distinction between data and instructions provided by M_{i+1} . The points of conversion from data to instructions are necessary, but they must be closely evaluated when reasoning about security.

3.1.2 Applications versus Data Files

In many operating systems, running applications and opening documents are both accomplished by double-clicking [10]. Double-clicking an application causes the operating system to convert data in the application into instructions in a new running process. Double-clicking a document causes the operating system to instruct some process to open the document as data. The overloaded meaning of double-clicking is a leaky abstraction that actively encourages users to not distinguish between instructions and data. The lack of distinction between applications and documents has enabled a wide range of social-engineering attacks where a user thinks he is opening a data file, but is actually launching a malicious application.

3.1.2.1 File Permissions and Mount Options

On UNIX-like operating systems, file permissions include *execute* permissions. By default, newly created files do not have execute permissions enabled. Files without execute permissions are not automatically converted from data to instructions.

Many file systems on UNIX-like operating systems can be mounted with a *noexec* option that disallows execution (i.e., conversion to instructions) of all files in that file system.

3.1.3 Processor Instructions versus Data

The von Neumann architecture consists of a central processing unit (CPU), a random access memory (RAM) that is divided into fixed size words, and input and output facilities. Instructions and data are both stored in the RAM. Most general-purpose computers that people are familiar with are based on the von Neumann architecture. For example, Intel x86, AMD64, and PowerPC are all von Neumann architectures. Since instructions and data are stored in the same RAM, many von Neumann architectures fail to distinguish between instructions and data. Without a distinction between instructions and data, adversaries can load data into RAM and cause the CPU to use the adversary-supplied data as instructions [11].

The Harvard architecture consists of a CPU, two RAMs that are each divided into fixed size words, and input and output facilities. One RAM is called the *instruction memory*, and the other RAM is called the *data memory*. Each RAM is in a separate address space and the CPU only executes instructions from the instruction memory. Many microcontrollers and digital signal processing systems are Harvard architectures.

3.1.3.1 $\neg W \vee \neg X$ Memory Protection

Not writable or not executable memory, written as $\neg W \vee \neg X$ (alternately, $\overline{W+X}$), means each value in memory can be writable or executable, but not both. Data can be writable but should not be executable. Instructions should be executable but should not be writable. For a pure Harvard architecture, this is accomplished by partitioning the memory hierarchy into a read-only instruction memory and a read-write data memory. The Harvard architecture hardware only fetches and executes instructions from the instruction memory. The two separate memories of the Harvard architecture are an inherent security solution for distinguishing between instructions and data on that machine. For a von Neumann architecture, this means that any specific symbol in memory should be exclusively writable or executable. A check in hardware or software is required before using any symbol in memory as either writable data or executable instruction and is therefore a programmatic security solution for distinguishing between instructions and data on that machine⁵.

Hardware and operating system support for $\neg W \vee \neg X$ memory protection effectively addresses the instruction versus data abstraction for CPU-native applications by making the reality provided by the machine match the abstraction used by programmers. Various publications point out that adversaries can still exploit systems with $\neg W \vee \neg X$ memory protection [13] [14] [15]. It is, however, important to note that the adversaries are not directly violating the instruction versus data separation; the adversaries are using different indirections or are using an insufficiently protected data-to-instructions conversion point.

⁵ AMD calls this feature in their processors a *No eXecute* (NX) bit. Intel calls this feature in their processors an *eXecute Disable* (XD) bit. OpenBSD support is called *W^X* (i.e., exclusively writable or executable). Linux support is part of *ExecShield* [12]. Microsoft Windows support is called *Data Execution Prevention* (DEP).

3.1.3.2 Instruction Set Randomization

Some work has been done on Instruction Set Randomization (ISR) as a defense against code injection attacks [16] [17]. An instruction set can be described as a static mapping between binary values and CPU operations such as load, store, add, subtract, multiply, and divide. ISR randomizes this mapping so that the adversary is faced with the (hopefully) computationally complex task of discovering a mapping from adversary-supplied data to valid instructions. ISR is a probabilistic security solution for distinguishing between instructions and data on that machine. Several attacks against ISR techniques are presented in [18] that reduce the complexity of discovering the data to instruction mapping used in [17]. However, alternate ISR approaches could provide a stronger randomized mapping.

3.1.4 SQL Instructions versus Data

The Structured Query Language (SQL) is commonly used in relational database systems. When creating applications that interface with a database, many programmers will dynamically build SQL commands using user-supplied data. In an SQL injection attack, an adversary includes SQL syntax in the user-supplied data. The adversary's SQL is executed when the dynamically built command is sent to the database [19].

3.1.4.1 Prepared SQL Statements

Parsing an SQL statement is the point of conversion from data to instructions. For most databases, once an SQL statement with placeholders for user-supplied parameters has been parsed and prepared, any parameters later supplied when executing that prepared statement are treated as data and do not pass through the parser. Since there are distinct instruction creation and data handling steps, prepared SQL statements provide an inherent security solution for distinguishing between instructions and data on that machine and preventing SQL injection attacks.

3.1.4.2 SQL Input Filtering

User-supplied data can be checked for SQL key words and escape characters before dynamically building an SQL command. Checking the user-supplied data is a programmatic security solution for distinguishing between instructions and data on that machine. However, due to the complexity of SQL syntax, support for various character encodings, and the interfaces between various databases and programming languages, properly checking the user-supplied data is error-prone and SQL injection attacks frequently slip past programmatic checks.

3.1.4.3 SQL Randomization

Randomizing SQL key words (i.e., instructions) has been demonstrated as a defense against SQL injection attacks [20]. Again, randomization increased the computational complexity for the adversary to discover a mapping from adversary-supplied data to valid SQL instructions. SQL randomization provides a probabilistic security solution for distinguishing between instructions and data on that machine and preventing SQL injection attacks.

3.1.5 JavaScript Instructions versus HTML Data

The Hyper-Text Markup Language (HTML) was originally designed as a markup language for data. The introduction of JavaScript added instructions to HTML. Example 3.1 is a simple example of an HTML document that contains both legitimate JavaScript that is part of the design and malicious JavaScript that an adversary placed in a product review or some other user-submitted data. When a web browser retrieves this HTML document, there is very little to distinguish between the legitimate and malicious JavaScript in the document.

```
<html>
  <head>
    <script type="text/javascript" src="external_file.js"></script>
    <script type="text/javascript">
      // Legitimate site-supplied JavaScript.
    </script>
  </head>
  <body>
    <!-- HTML data goes here. -->
    ...
    <script type="text/javascript">
      // Legitimate site-supplied JavaScript.
    </script>
    ...
    <!-- Server-generated content: such as user comments. -->
    <!-- Adversary submitted script tag in a comment field. -->
    <script type="text/javascript">
      // Malicious adversary-supplied JavaScript.
    </script>
    ...
  </body>
</html>
```

Example 3.1 Vulnerable HTML document.

3.1.5.1 NoScript Firefox Add-On

NoScript [21] is an add-on for the Firefox web browser that disables execution of all JavaScript by default. The user then explicitly allows execution of JavaScript that is downloaded from web sites that the user trusts. If you view the Internet as a distributed memory, NoScript enables partitioning the Internet into separate instruction and data memories.

3.1.5.2 Create Separate Code and Data Sections in HTML

We propose that a *noexec* attribute be added to the HTML specification. The *noexec* attribute would instruct the browser to disable JavaScript parsing and execution on the document sub-tree within any element that has the *noexec* attribute. For documents that don't use JavaScript, the document authors could apply the *noexec* attribute to the `<html>` element to prevent all JavaScript execution. For documents that only contain JavaScript in the `<head>` element, the document authors could apply the *noexec* attribute to the `<body>` element. For documents that have legitimate JavaScript in the `<body>` element, the document authors could apply the *noexec* attribute to any `<div>` or similar element that might contain user-supplied data.

Web browser support for a noexec attribute would provide a way for legitimate sites to protect their users against adversary-supplied JavaScript by specifying the author-intended distinction between instructions and data in the document. Support for a noexec attribute would not protect users against malicious sites that intentionally mark their adversary-supplied JavaScript as executable. This parallels $\neg W \vee \neg X$ memory protection that prevents adversary-supplied instructions from being injected into legitimate applications but does not prevent users from executing malicious applications.

3.2 Pointers

Pointers are an excellent example of indirection. Pointers are data, but they can contain the addresses of either instructions or data. Classic stack-smashing buffer overflows [11], return-into-libc exploits [13], and return-oriented programming [22] are some of the vulnerability classes that rely on manipulating pointers.

3.2.1 Remove Pointer Access

Haskell, Java, Python, and Ruby are examples of programming languages that remove pointers from the programmer's direct control to provide an inherent security solution against pointer manipulation for programs written in those languages. It must be noted that the language interpreters or compiled executables still use pointers internally and that defects in the interpreter or compiler may generate pointer vulnerabilities.

3.2.2 Pointer Value Checks

If there is a specific range or set of addresses that can be defined as the valid region for a pointer to reference, programmatic checks can be added at each pointer dereference to ensure the pointer's value is in the valid region. Pointer value checks are a programmatic security solution for pointer manipulation.

3.2.3 Address Space Layout Randomization

Address space layout randomization (ASLR) randomizes where in the address space of each new process various program sections are located. ASLR is a probabilistic security solution for pointer manipulation because it creates a randomized mapping between memory content that an adversary wants to access (such as libraries, the stack, and items on the heap) and memory addresses (i.e., pointers). However, on 32-bit platforms, the number of bits that can be randomized in ASLR is limited to $k < 14$, allowing brute force discovery of the randomized mapping [23]. On 64-bit platforms, k can be larger, making brute force discovery less feasible.

3.2.4 Pointer Encryption

PointGuard [24] modifies the compiler to generate additional object code that encrypts pointers as they are moved from registers to memory and decrypts pointers as they are moved from memory to registers. The encryption and decryption are done in the registers, so the decrypted pointers are never present in memory. Pointer encryption is a probabilistic security solution for pointer manipulation because it creates a mapping between pointers in memory and pointers in registers that is computationally complex for an adversary to discover. If an adversary modifies an encrypted pointer in memory, the value of the decrypted pointer in a register is not predictable for the adversary.

3.3 Data Structures versus Unstructured Memory

Computers have unstructured memory corresponding to the tape in a universal Turing machine. A data structure⁶ is an abstraction that imposes structure and meaning on a region of otherwise unstructured memory. In a language like C, the abstraction mismatch is that the programmer is designing in terms of data structures, but the interfaces to those data structures are through direct access to the underlying memory. In particular, C arrays and structs are data structures, but access to them is through pointer arithmetic and memory access functions such as `memcpy`, `strcpy`, `strncpy`, and `gets`, that have no inherent knowledge of data structures. It is the programmer's (often error-prone) job to manage the mapping between these layers of abstraction. When the programmer fails to properly manage the mapping, vulnerabilities are introduced into the system. From this perspective, there are two basic techniques that can be applied to reduce the abstraction mismatch between data structures and unstructured memory: remove access to unstructured memory or add a run-time enforced structure to memory.

3.3.1 Remove Unstructured Memory Access

Object-oriented languages provide complex data structures but also provide interfaces to those data structures that are at a higher layer of abstraction and contain knowledge of the data structures being accessed. Several object-oriented languages, such as Java, remove the option of unstructured memory access from the programmer, providing an inherent security solution for the vulnerability-prone mapping of data structures into unstructured memory. In the C++ programming language, the programmer can write object-oriented classes but still retains access to unstructured memory. C++ class interfaces can include checks that encode knowledge about data structures to provide a partial programmatic security solution for mapping data structures into unstructured memory. The reduced mismatch between data structures and the methods of accessing them is the reason object-oriented languages are less prone to memory access errors than C.

3.3.2 Stack Smashing Protection

A contrived program is presented in Example 3.2 that includes multiple buffer overflow vulnerabilities. This example will allow us to discuss buffer overflows and some of the current defenses against buffer overflows. When the code from Example 3.2 is compiled and `func()` is called, the top of the stack will be structured as shown in Figure 3.1.

After the publication of [11], attacks using buffer overflows became common. StackGuard [25] introduced a way of detecting and stopping stack-smashing buffer overflows by placing a special value called a *canary*⁷ above the saved instruction pointer on the stack. The value of the canary is checked immediately before the function returns. If the canary is unchanged, the function returns normally. If the canary has been altered, the program will abort without returning from the function. However, the saved frame pointer is not protected by the canary with StackGuard version 1 or 2, which allows an adversary to bypass the StackGuard protections without overwriting the canary [26]. Based on the ordering of local variables in the function, local pointer variables could also be overwritten without overwriting the canary, which enables various multi-step attacks [27].

⁶ Examples of data structures include arrays, linked lists, stacks, queues, heaps, trees, and graphs.

⁷ The name *canary* came from the canary birds that were used by miners to detect if the air in a mine shaft was becoming toxic. If the canary died, the miners would evacuate the mine shaft.

```

#include <stdio.h>
#include <stdlib.h>

int func(int arg_data_variable,
        int *arg_data_pointer,
        int (*arg_func_pointer)(int))
{
    int data_variable = 7;
    int (*func_pointer)(int*);
    char *buffer0 = malloc(64);
    char buffer1[64];

    gets(buffer0);
    gets(buffer1);

    func_pointer(arg_data_pointer);
    arg_func_pointer(data_variable);

    return (0);
}

int main (int argc, char* argv[])
{
    return (func(5, NULL, NULL));
}

```

Example 3.2 Vulnerable C program.

Figure 3.1 i386 stack: Unprotected.

ProPolice [28] is based on ideas from StackGuard, but incorporates a *safe frame structure* model. The safe frame structure model moves the location of the canary on the stack to protect all saved registers (including the saved instruction and frame pointers) and re-orders the function’s local variables so that any arrays and array-containing structures are placed on the stack below non-buffer local variables. In the safe frame structure model, buffer writes move away from the non-buffer local variables and toward the stack canary. To protect function pointer arguments from being overwritten and later dereferenced within the function, the function pointer arguments are copied into the non-buffer local variable section and these safe copies

of the function pointer arguments are dereferenced within the function instead of dereferencing the actual arguments.

Figure 3.2 i386 stack: ProPolice.

The result of applying ProPolice’s safe frame structure is shown in Figure 3.2. If a buffer overflow occurs, the overflow is writing in the direction away from the non-buffer local variables (i.e., protected_a in Figure 3.2), preventing the non-buffer local variables from being overwritten. The saved instruction pointer, saved frame pointer, and other saved registers (i.e., protected_b in Figure 3.2) are not used until the function returns and are, therefore, protected by the canary. The basic ProPolice model has been adopted by several compilers and is available through GCC’s `-fstack-protector` or `-fstack-protector-all` options and Microsoft Visual Studio’s `/GS` option.

Stack smashing protections are a good example of how focusing on the indirections involved can enable better security solutions. StackGuard originally focused on protecting the return address. ProPolice focused on protecting as much of the stack as possible. The evolution of stack smashing protections have incrementally added an enforced stack frame structure to unstructured memory. By reducing the abstraction mismatch between unstructured memory and data structures, buffer overflows have been prevented that would have otherwise been possible.

However, certain buffer overflows are still possible because stack smashing protections have only reduced, and not eliminated, this abstraction mismatch. For example, heap overflows [29] are not prevented. Format string vulnerabilities [30] can index into the stack, skipping over the canary, and overwrite the saved instruction pointer or other stack values. An overflow of `buffer1` can overwrite the function arguments passed on the stack (i.e., unprotected_b in Figure 3.2). This overflow will corrupt the canary, but the overwritten function arguments may be used in the function, to the adversary’s advantage, before the canary is checked [26]. Any objects with both data buffers and virtual member functions group together data and a pointer to a table of function pointers, which can limit the effectiveness of the safe frame structure when such objects are on the stack.

3.4 Integer Errors

At a young age, humans learn to use the countably infinite set of non-negative integers $\mathbb{N}$ and the countably infinite set of integers $\mathbb{Z}$ where:

$$\begin{aligned}\mathbb{N} &= \{0, 1, 2, 3, \dots\} \\ \mathbb{Z} &= \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}\end{aligned}$$

Unfortunately, the basic integer types supported by processors can only represent a finite number of integer values.

A k -bit unsigned integer can only represent the finite set of non-negative integers

$$\mathbb{U}^k = \{0, 1, \dots, 2^k - 1\}$$

A k -bit, two's-complement, signed integer can only represent the finite set of integers

$$\mathbb{S}^k = \{-2^{k-1}, \dots, -1, 0, 1, \dots, 2^{k-1} - 1\}$$

The C99 integer types defined in the `<stdint.h>` header file provide several concrete examples of $\mathbb{U}^k$ and $\mathbb{S}^k$ that are available in common processors: `uint8_t` is $\mathbb{U}^8$, `uint16_t` is $\mathbb{U}^{16}$, `uint32_t` is $\mathbb{U}^{32}$, `uint64_t` is $\mathbb{U}^{64}$, `int8_t` is $\mathbb{S}^8$, `int16_t` is $\mathbb{S}^{16}$, `int32_t` is $\mathbb{S}^{32}$, and `int64_t` is $\mathbb{S}^{64}$.

The abstraction mismatch between human programmers thinking in $\mathbb{N}$ or $\mathbb{Z}$ and the processor operating on $\mathbb{U}^k$ or $\mathbb{S}^k$ is the source of a class of problems known as *integer errors*.

When considering the relationships between these sets:

$$\begin{aligned}\mathbb{U}^k &\subset \mathbb{U}^{k+n} \subset \mathbb{N} \quad \text{for all } n > 0 \\ \mathbb{S}^k &\subset \mathbb{S}^{k+n} \subset \mathbb{Z} \quad \text{for all } n > 0 \\ \mathbb{N} &\subset \mathbb{Z} \\ \mathbb{U}^k &\not\subset \mathbb{S}^k \quad \text{because } \{2^{k-1}, \dots, 2^k - 1\} \not\subset \mathbb{S}^k \\ \mathbb{U}^k &\subset \mathbb{S}^{k+1}\end{aligned}$$

Table 3.1 lists the requirements to accurately represent binary operations over these finite sets.

Table 3.1 Integer operations.

Binary Operations	Required for Correctness	Implemented by Processors
unsigned addition	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{U}^{k+1}$	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{U}^k$
signed addition	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^{k+1}$	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^k$
unsigned subtraction	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{S}^{k+1}$	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{U}^k$
signed subtraction	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^{k+1}$	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^k$
unsigned multiplication	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{U}^{2k}$	$\mathbb{U}^k \times \mathbb{U}^k \rightarrow \mathbb{U}^k$
signed multiplication	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^{2k}$	$\mathbb{S}^k \times \mathbb{S}^k \rightarrow \mathbb{S}^k$

Note that in the correct form for each of these binary functions, the domain is a proper subset of the codomain. Whenever a binary function produces a result in the codomain that is not in the domain, an integer overflow error will occur. Integer sign errors occur when converting in either direction between $\mathbb{U}^k$ and $\mathbb{S}^k$ if the

intended value is not in both sets. Integer truncation errors occur if the intended value is not in both sets when converting from $\mathbb{U}^{k+n} \rightarrow \mathbb{U}^k$ or when converting from $\mathbb{S}^{k+n} \rightarrow \mathbb{S}^k$ for any $n > 0$.

Integer errors can contribute to vulnerabilities by causing incorrect decisions to be made based on the results of the integer error or by enabling incorrectly bounded access to memory [31] [32]. There are three methods of processing an integer error [33]:

1. Strict interpretation: an operation that results in an integer error has a non-existent result and the offending program terminates.
2. Firm boundary (saturation): an operation that results in an integer error has a result that is the appropriate maximum or minimum representable value.
3. Modulo arithmetic: an operation that results in an integer error has a result that is computed modulo the size of the set of representable integers.

Most processors perform modulo arithmetic and set flags in a status register to indicate if integer errors may have occurred. Either a strict interpretation or a firm boundary can be emulated by adding instructions that check the appropriate flags and conditionally raise an error or set the result to the boundary value.

3.4.1 Multiple-Precision Integers

On a k -bit processor, multiple-precision integers are implemented with multiple k -bit words in memory and algorithms to perform the required sequences of operations over those multiple words. The classical multiple-precision arithmetic algorithms are presented in [34]. Multiple-precision integers are an inherent security solution implemented by programmatic checks in another layer of abstraction.

3.4.2 Run-Time Error Detection

The integer error detection and protection in [35] address the programmer's flawed abstractions by modifying the compiler to generate additional object code that checks the processor's status flag after each binary integer operation or integer type conversion. The strict interpretation is implemented by terminating the program if the compiler-generated checks detect an integer error. The compiler-generated checks in [35] provide a programmatic security solution to the class of integer errors.

GCC's `-ftrapv` option and Microsoft Visual Studio's `/RTCc` option each instruct the compiler to generate certain limited run-time checks for integer errors [36]. Similarly, the `boost::numeric_cast` C++ library [37] throws an exception on integer truncation or sign conversion errors that occur when `boost::numeric_cast` is used to explicitly perform the conversion.

3.5 The Network Stack

The Open Systems Interconnection (OSI) reference model depicted in Figure 3.3 contains multiple layers of indirection in the network stack.

Figure 3.3 OSI reference model.

In Figure 3.3, solid arrows represent direct communication and dotted arrows represent indirect communication. Each layer of the OSI reference model directly provides functionality to the layer above it, directly relies on functionality provided by the layer below it, and indirectly communicates with its peer layer on another system. Ethernet is the most common data link layer encountered in an office environment today. The Internet Protocol (IP) version 4 (IPv4) [38] is the standard network layer in use today, and IP version 6 (IPv6) [39] is starting to be deployed as the next generation network layer.

The mappings between the OSI layers are popular targets for adversaries.

3.5.1 CAM Table Flooding

The content addressable memory (CAM) in an Ethernet switch implements a mapping between Ethernet Media Access Control (MAC) addresses and physical ports on the switch. Switches are designed primarily to boost network performance, but there is also an improvement in network confidentiality since Ethernet frames are no longer broadcast to every device in the Ethernet network. Ethernet frames are only sent out the switch port associated with the Ethernet destination MAC address in the switch's CAM table.

However, CAM tables can hold a finite number of mappings. An adversary can spoof a large number of Ethernet MAC addresses on a switch port to flood the switch's CAM table with more entries than it can hold. Once the switch's CAM table is full, new valid mappings can not be stored. When a destination Ethernet MAC address is not found in the CAM table, the switch broadcasts the Ethernet frame out all ports. The adversary can then observe the broadcast frames.

3.5.1.1 Restricted Mappings Between Ethernet MAC Addresses and Switch Ports

Many business-grade switches can be configured with static mappings between Ethernet MAC addresses and switch ports. Alternately, many business-grade switches can specify a limit on the number of Ethernet MAC addresses mapped to each switch port. If an unauthorized mapping is encountered or the number of allowed mappings is exceeded, the offending switch port is disabled. Both approaches are programmatic security solutions for protecting against CAM table flooding because the switch performs run-time checks as Ethernet frames enter ports on the switch.

3.5.2 ARP and NDP Redirection

The Address Resolution Protocol (ARP) [40] is used to build a mapping between Ethernet MAC addresses and IPv4 network addresses. Similarly, the Neighbor Discovery Protocol (NDP) [41] is used to build a mapping between Ethernet MAC addresses and IPv6 network addresses. By influencing the mapping between Ethernet addresses and IP addresses, an adversary can compromise traffic between systems on that Ethernet network.

Let hosts A , B , and C be part of the same Ethernet network. Host A has IP network address IP_A and Ethernet address MAC_A . Host B has IP network address IP_B and Ethernet address MAC_B . Host C has Ethernet address MAC_C . When hosts A and B need to communicate, the ARP or NDP is used to establish a mapping from IP_A to MAC_A on host B and to establish a mapping from IP_B to MAC_B on host A . An adversary controlling host C can manipulate ARP or NDP so that the incorrect mapping from IP_A to MAC_C exists on host B and the incorrect mapping from IP_B to MAC_C exists on host A . All IP traffic between hosts A and B will then be sent over Ethernet to the adversary-controlled host C where the traffic can be monitored, modified, or discarded by the adversary.

The fact that an adversary can influence the dynamically established mapping between IP addresses and Ethernet MAC addresses is the reason that ARP and NDP redirection is possible.

3.5.2.1 Static Mappings Between Ethernet MAC Addresses and IP Addresses

System owners can enter static entries into the ARP cache. Since the static ARP cache entries are not dynamically established and do not expire, adversaries can't control that mapping between Ethernet MAC address and IP address. Static ARP cache entries are an inherent security solution for protecting the mapping between Ethernet MAC addresses and IPv4 addresses. The downside of static ARP caches, and why they are infrequently used, is that maintaining the set of static mappings on every system is a burden on the network administrators.

IPv6 addresses are 128 bits long. The current subnet definition is the most significant 64-bits, leaving the least significant 64-bits for the local identifier. One common method for dynamically obtaining an IPv6 address is to derive the local identifier portion of the IPv6 address from the 48-bit Ethernet MAC address using the process specified in [42]. This allows a simple new defense against NDP spoofing in any network that has a policy of only allowing IPv6 addresses that are derived from Ethernet MAC addresses. Systems simply verify that the Ethernet MAC address and the IPv6 address in any NDP messages are consistent with the EUI-64 based mapping specified in [42]. If the Ethernet and IPv6 addresses are not consistent with the expected mapping function, then a system is either using a different algorithm for obtaining an IPv6 address (violating policy) or NDP spoofing is occurring (violating policy). Systems can ignore any NDP messages that are not consistent with the expected mapping function to achieve the security benefits of static ARP tables without the administrative overhead. This expected mapping function is a programmatic security solution for protecting the mapping between Ethernet MAC addresses and IPv6 addresses.

3.5.2.2 Cryptographic Mappings Between Ethernet MAC Addresses and IP Addresses

Secure Neighbor Discovery (SEND) [43] relies on cryptography and a set of certification authorities to verify that NDP messages originated from certified routers and to verify the integrity of the NDP messages. SEND is a probabilistic security solution for protecting the mapping between Ethernet MAC addresses and IPv6 addresses.

3.5.3 DNS Redirection

The Domain Name System (DNS) [44] provides a mapping between human-friendly host names and IP network addresses. Host A has the IP network address IP_A . Host B has the name $Name_B$ and the IP network address IP_B . When host A has a message to send to host B , host A will use the DNS to establish a mapping from $Name_B$ to IP_B . An adversary controlling host C with IP network address IP_C can manipulate the DNS so that the incorrect mapping from $Name_B$ to IP_C exists on host A . When host A tries to communicate with host B by using $Name_B$, host A will actually be communicating with the adversary-controlled host C at IP_C .

DNS redirection is very similar to ARP redirection. While the technical details of their implementation differ, DNS redirection and ARP redirection are examples of the same vulnerability at different layers of indirection.

3.5.3.1 Static Mappings Between Host Names and IP Addresses

Static DNS entries are a direct parallel to static ARP cache entries. Eliminating the dynamic establishment of the mapping provides an inherent security solution for protecting the mapping between host names and IP addresses. However, like static ARP cache entries, static DNS entries are infrequently used due to the burden of managing the static mappings that is placed on the network administrators.

3.5.3.2 Cryptographic Mappings Between Host Names and IP Addresses

DNS Security Extensions (DNSSEC) [45] rely on cryptography and a set of certification authorities. With DNSSEC, DNS answers are digitally signed so the receiving host can verify that the answer originated from a certified DNS server and can verify the integrity of the answer. DNSSEC is a probabilistic security solution for protecting the mappings between host names and IP addresses.

3.5.4 Security Solutions that Cross Layers of Indirection

Each of the network security solutions described in this section span two or more layers of the network stack. An additional example is how firewall rules typically specify combinations of properties from OSI layer 4 (i.e., TCP or UDP source and destination port numbers) and OSI layer 3 (i.e., IPv4 or IPv6 source and destination addresses). Security solutions that span multiple layers of indirection provide the context necessary to protect some of the mappings between layers of the network stack.

3.6 The Hardware and Software Stack

The hardware and software stack depicted in Figure 3.4 is a graphical representation of an abstract machine hierarchy as described in Section 1.1.1 and a reasonable approximation of modern personal computers. The operational environment of each layer is defined by the layers below it in the abstract machine hierarchy.

Historically, adversaries have focused on the higher layers of the hardware and software stack (e.g., viruses at the application layer, Trojan horses at the system services and operating system layers, and rootkits at the operating system layer). But in recent years, there has been a focus on finding vulnerabilities in the lower layers of the stack. As vulnerabilities are mitigated in M_i , adversaries are targeting M_{i-1} .

Figure 3.4 Hardware/Software stack.

3.6.1 Operating System Rootkits

The Interrupt Descriptor Table (IDT) provides a mapping between hardware events and event handlers in the OS. IDT hooking modifies this mapping to execute malware. The System Services Descriptor Table (SSDT) or System Call (syscall) table provides a mapping between user-space system calls and functionality in the OS. SSDT hooking or syscall hooking modifies this mapping to execute malware whenever the hooked system call is invoked.

3.6.2 Malicious Virtual Machine Managers

Historically, operating systems have been designed to run directly on the machine defined by the hardware and firmware. The introduction of virtual machine manager technology has added a layer of indirection between the operating system and the machine defined by the hardware and firmware. Multiple publications have described how a malicious virtual machine manager can be used to subvert the security of the operating system [46] [47] [48].

3.6.3 Malicious Firmware

Various publications have described how modifications to boot-related firmware can be used to bootstrap malicious functionality before the OS loads [49] [50] [51].

3.6.4 Malicious Hardware

A maliciously-modified CPU is demonstrated in [52]. First, the modified CPU allows unprivileged software to access privileged memory regions via the addition of 959 logic gates that disable memory protections when a special sequence of data traverses the data bus. Second, the modified CPU provides a new shadow execution mode via the addition of 1,341 logic gates that reserve instruction and data cache lines for malware and trap events from normal mode into shadow mode. The unmodified CPU had 1,787,958 logic gates, so the addition of both malicious features increased the size of the design by only 0.13%.

3.6.5 Security Solutions that Cross Layers of Indirection

The Trusted Platform Module (TPM) provides a hardware-based random number generator, protected generation and storage of cryptographic keys, *attestation* of boot process integrity, and *sealing* (i.e., encryption) of data based on attestation values. Attestation involves calculating a cryptographic hash of boot-related firmware, boot loaders, and the OS. The resulting cryptographic hashes for each stage of the boot process are compared against trusted values before transferring execution to that stage. The chain of cryptographic hashes in attestation spans multiple layers of indirection and provides a probabilistic security solution for protecting some of the mappings between layers of the software stack.

4 Conclusion

We proposed that focusing on the indirections in system designs is a better way to design, evaluate, and compare security solutions. We demonstrated how a wide variety of common vulnerabilities can be analyzed using this approach and how vulnerabilities at different locations in the abstract machine hierarchy can be grouped together into vulnerability classes. By translating security solutions across vulnerabilities in the same class, we were able to propose new security solutions to defend against NDP redirections and JavaScript injected into HTML documents.

Our method also simplifies the comparison of security solutions. If two security solutions address different indirections, then the two security solutions are complementary. If two security solutions address the same indirection, then the two security solutions can be compared by whether each is an inherent, programmatic, or probabilistic security solution and by how completely each security solution deals with the underlying indirection.

We proved that certain classes of security solutions, such as enforcing a distinction between instructions and data, do not, in general, propagate across abstract machine boundaries. The implication is that each abstract machine in an abstract machine hierarchy must implement its own security solutions to defend against certain classes of vulnerabilities.

We expect our method to be useful in developing security metrics and requirements. Example security metrics include:

- What percentage of systems enforce a distinction between instructions and data?
- What percentage of systems protect against unstructured memory access?

Design review or procurement requirements could require a list of the Turing-equivalent abstract machines in the design and a list of the security solutions implemented in each abstract machine. We will explore such metrics and requirements in a future report.

References

- [1] D. Spinellis, *Another level of indirection*, in *Beautiful Code: Leading Programmers Explain How They Think*, edited by A. Oram and G. Wilson (O'Reilly Media, Inc., 2007), pp. 279–291.
- [2] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. (Addison-Wesley, 1979).
- [3] L. Robinson and K. N. Levitt, *Proof techniques for hierarchically structured programs*, in *Communications of the ACM* (ACM, 1977), pp. 271–283.
- [4] S. Engle and M. Bishop, *A model for vulnerability analysis and classification* Tech. Rep. CSE-2008-5, UC Davis (2008).
- [5] M. Torgerson *Security metrics for communication systems*, in *Proceedings of the 12th International Command and Control Research and Technology Symposium* (DoD CCRP, 2007).
- [6] S. Engle, S. Whalen, and M. Bishop, *Modeling computer insecurity* Tech. Rep. CSE-2008-14, UC Davis (2008).
- [7] Merriam-Webster, Inc., *Merriam-Webster's Collegiate Dictionary*. (Merriam-Webster, Inc., 2008), 11th ed.
- [8] R. Pucella and F. B. Schneider, *Independence from obfuscation: A semantic framework for diversity* Tech. Rep. TR2006-2016, Cornell University (2006).
- [9] J. Spolsky, *The law of leaky abstractions* (2002).
- [10] K. Yee, *Guidelines and strategies for secure interaction design*, in *Security and Usability: Designing Secure Systems that People Can Use*, edited by L. F. Cranor and S. Garfinkel (O'Reilly Media, Inc., 2005), pp. 247–274.
- [11] Aleph One, *Smashing the stack for fun and profit*, in *Phrack* (Phrack, 1996).
- [12] U. Drepper, *Security enhancements in red hat enterprise linux: (besides SELinux)* (2005)
- [13] Nergal, *The advanced return-into-lib(c) exploits*, in *Phrack* (Phrack, 2001).
- [14] H. Shackam *The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)*, in *Proceedings of the 14th ACM Conference on Computer and Communications Security* (ACM, 2007), pp. 552–561.
- [15] D. Blazakis, *Interpreter exploitation: Pointer inference and JIT spraying* (2010)
- [16] G. S. Kc, A. D. Keromytis, and V. Prevelakis *Countering code-injection attacks with instruction-set randomization*, in *Proceedings of the 10th ACM Conference on Computer and Communications Security* (ACM, 2003), pp. 272–280.
- [17] E. G. Barrantes et al. *Randomized instruction set emulation to disrupt binary code injection attacks*, in *Proceedings of the 10th ACM Conference on Computer and Communications Security* (ACM, 2003), pp. 281–289.
- [18] A. N. Sovarel, D. Evans, and N. Paul *Where's the FEEB?: The effectiveness of instruction set randomization*, in *Proceedings of the 14th USENIX Security Symposium* (USENIX Association, 2005), pp. 145–160.
- [19] rain.forest.puppy, *NT web technology vulnerabilities*, in *Phrack* (Phrack, 1998).
- [20] S. W. Boyd and A. D. Keromytis, *SQLrand: Preventing SQL injection attacks* (2004)
- [21] G. Maone, *NoScript*.
- [22] S. Checkoway et al. *Return-oriented programming without returns*, in *Proceedings of the 17th ACM Conference on Computer and Communications Security* (ACM, 2010), pp. 559–572.
- [23] H. Shacham et al. *On the effectiveness of address-space randomization*, in *Proceedings of the 11th ACM Conference on Computer and Communications Security* (ACM, 2004), pp. 298–307.

- [24] C. Cowan, S. Beattie, J. Johansen, and P. Wagle *PointGuard: Protecting pointers from buffer overflow vulnerabilities*, in *Proceedings of the 12th USENIX Security Symposium* (USENIX Association, 2003), pp. 91–104.
- [25] C. Cowan et al. *StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks*, in *Proceedings of the 7th USENIX Security Symposium* (USENIX Association, 1998), pp. 63–78.
- [26] G. Richarte, *Four different tricks to bypass StackShield and StackGuard protection* Tech. Rep., Core Security Technologies (2002).
- [27] Bulba and Kil3r, *Bypassing stackguard and stackshield*, in *Phrack* (Phrack, 2000).
- [28] H. Etoh, *announcement of machine independent stack protection code* (2000)
- [29] F. Lindner, *A heap of risk: Buffer overflows on the heap and how they are exploited* (2006).
- [30] gera and riq, *Advances in format string exploitation*, in *Phrack* (Phrack, 2002).
- [31] blexim, *Basic integer overflows*, in *Phrack* (Phrack, 2002).
- [32] O. Horovitz, *Big loop integer protection*, in *Phrack* (Phrack, 2002).
- [33] C. W. Hoare, *An axiomatic basis for computer programming*, in *Communications of the ACM* (ACM, 1969), pp. 576–580.
- [34] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. (Addison-Wesley, 1998a), 3rd ed.
- [35] D. Brumley et al. *RICH: Automatically protecting against integer-based vulnerabilities*, in *Proceedings of the 14th Network & Distributed System Security Symposium* (Internet Society, 2007).
- [36] R. C. Seacord, *Secure Coding in C and C++*. (Addison-Wesley, 2005).
- [37] F. Carballal, *Boost numeric_cast*.
- [38] Information Sciences Institute University of Southern California *Internet Protocol*. (1981).
- [39] S. Deering and R. Hinden *Internet Protocol, Version 6 (IPv6) Specification*. (1998).
- [40] D. C. Plummer *An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48-bit Ethernet Addresses for Transmission on Ethernet Hardware*. (1982).
- [41] T. Narten, E. Nordmark, W. Simpson, and H. Soliman *Neighbor Discovery for IP version 6 (IPv6)*. (2007).
- [42] R. Hinden and S. Deering *IP Version 6 Addressing Architecture*. (1998).
- [43] J. Arkko, J. Kempf, B. Zill, and P. Nikander *SEcure Neighbor Discovery (SEND)*. (2005).
- [44] P. Mockapetris *Domain Names - Implementation and Specification*. (1987).
- [45] R. Arends et al. *DNS Security Introduction and Requirements*. (2005).
- [46] S. T. King et al. *SubVirt: Implementing malware with virtual machines*, in *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (IEEE, 2006), pp. 314–327.
- [47] D. A. Dai Zovi *Hardware virtualization rootkits*, in *Proceedings of Black Hat USA 2006* (Black Hat, 2006).
- [48] J. Rutkowska *Subverting Vista kernel for fun and profit*, in *Proceedings of Black Hat USA 2006* (Black Hat, 2006).
- [49] J. Heasman *Implementing and detecting an ACPI BIOS rootkit*, in *Proceedings of Black Hat Federal 2006* (Black Hat, 2006a).
- [50] J. Heasman, *Implementing and detecting a PCI rootkit* (2006b)
- [51] J. Heasman *Hacking the extensible firmware interface*, in *Proceedings of Black Hat USA 2007* (Black Hat, 2007).
- [52] S. T. King et al. *Designing and implementing malicious hardware*, in *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats* (USENIX Association, 2008).

Distribution

- 1 MS 0123 D. Chavez, LDRD Office, 1011
- 1 MS 0899 Technical Library, 9536 (electronic copy)