Holographic Vulnerability Studies: Vulnerabilities as Fractures in Interpretation as Information Flows Across Abstraction Boundaries

Jedidiah R. Crandall
Univ. of New Mexico, Dept. of Computer Science
Mail Stop: MSC01 1130, 1 Univ. of New Mexico,
Albuquerque, NM 87131
crandall@cs.unm.edu

Daniela Oliveira
Bowdoin College, Dept. of Computer Science
8650 College Station, Brunswick, ME 04011
doliveir@bowdoin.edu

ABSTRACT

We are always patching our systems against specific instances of whatever the latest new, hot, trendy vulnerability type is. First it was time-of-check-to-time-of-use, then buffer overflows, then SQL injection, then cross-site scripting. Vulnerability studies are supposed to accomplish two main goals: to classify vulnerabilities into general classes so that unknown vulnerabilities of that class can be discovered in a proactive way, and to enable us to understand the fundamental nature of vulnerabilities so that when we build new systems we know how to make them secure. In this paper we propose a new paradigm for vulnerability studies: we view vulnerabilities as fractures in the interpretation of information as the information flows across the boundaries of different abstractions. We argue that categorizing vulnerabilities based on this view, as opposed to the types of categories that have been used in past vulnerability studies, makes vulnerability types more easily generalizable and avoids problems where vulnerabilities could be put in multiple categories.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Vulnerabilities, TOCTTOU, layers of abstraction, information flow.

1. INTRODUCTION

In his seminal paper on computer viruses [15], Cohen said, "*information only has meaning in that it is subject to interpretation*." This fact is at the crux of vulnerabilities in systems. As information flows from one process to another and influences the receiving process, interpretations of that information can lead to the receiving process doing things on the sending process' behalf that the system designer did not intend to allow as per the security model.

Consider buffer overflows [6]. At the level of abstraction of the application, an input is just a set of bytes that might be interpreted as a request formatted in a particular protocol. If some of the information goes beyond the bounds of a buffer at the compiler and architecture level, part of the information may be interpreted as control data. Attackers exploit this fact to corrupt the receiving process and cause it to execute machine code of their choosing.

Consider also SQL injection [27, 52]. Because the attacker puts escape characters in their inputs the inputs become not just leaves in the parse tree when the SQL query is generated, but part of the structure and semantics of the SQL query itself [27, 52]. It is as if the designers of the system intended a flat interpretation of the remote user's inputs, but an unexpected code path causes the user input to be fractured at a higher level of interpretation. Cross-site scripting [58] works in a similar way from this perspective.

Information, when viewed from the different perspectives for the various levels of abstraction that make up the system (OS, application, compiler, architecture, *etc.*), should still basically have the same interpretation. Fractures in interpretation as the information crosses abstraction boundaries are therefore precipitous changes in how the untrustworthy input can control the system, *i.e.*, probably vulnerabilities. We propose viewing vulnerabilities as fractured holograms. The Merriam Webster dictionary defines a hologram as, "*a three-dimensional image produced from a pattern of interference.*" Interference is the key word in our discussions about the origins of vulnerabilities. Interference can be seen as the way that untrusted inputs affect a system through information flow. Imagine a hologram of a rabbit. As the hologram is rotated and viewed from different perspectives, the image changes but is still an image of a rabbit. If the hologram, when viewed from a specific angle, becomes instead a picture of a cat, we can say the hologram is fractured. Similarly, data such as an HTTP request input should have different interpretations from the perspectives of the various levels of abstraction (as a string of bytes, as an HTTP request, partly as a filename, *etc.*). But when one level of abstraction interprets the information as a filename and another as a return pointer we can say that there has been a *precipitous* fracture in interpretation.

We propose a new paradigm that views vulnerabilities as fractures in interpretation as information flows. Specifically, at the boundaries between layers of abstraction there should always be differences in interpretation, but these differences should be relatively continuous (*e.g.*, a raw string of bytes *vs.* a string of bytes that is parsed as an HTTP GET request) rather than precipitous fractures (*e.g.*, a raw string of bytes *vs.* a string of bytes containing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NSPW'12, September 18–21, 2012, Bertinoro, Italy.
Copyright 2012 ACM 978-1-4503-1794-8/12/09 ...$15.00.

a mixture of an HTTP GET request, a return pointer, a stored stack pointer, and so forth).

Past vulnerability studies have suffered from two problems that we will focus on specifically in this paper. One problem is that vulnerability classifications are not easily generalizable, leading to the addition of many new classifications to cover new types of vulnerabilities. The second problem is that a single vulnerability type can fall under many different classifications, leading to ambiguity. In this paper we argue that viewing vulnerabilities as fractures in interpretation as information flows across abstraction boundaries leads to classifications that are more easily generalizable and less ambiguous.

This paper is structured as follows. First, in Section 2, we give concrete examples of fractures in interpretation for buffer overflows and SQL injection, and discuss the generality of viewing vulnerabilities in terms of information flow. Then, Section 3 gives a brief background of time-of-check-to-time-of-use vulnerabilities with a simple example that is used as a running example throughout the rest of the paper. This is followed by Section 4, which analyzes this type of vulnerability from the perspective of different levels of abstraction, and Section 5 which focuses on information flow. Then we conclude with related work and a list of questions we would like to pose to workshop attendees.

2. THE NATURE OF VULNERABILITIES

In this section we first make the view of vulnerabilities being fractures in interpretation as information flows across boundaries more concrete by giving more details on two vulnerability types mentioned in the Introduction (buffer overflows and SQL injection). Then we discuss the generalizability of this view for a broad range of vulnerabilities. Finally, we discuss how our new proposed paradigm for vulnerability studies is positioned with respect to common mitigation strategies, with a focus on memory corruption attacks.

2.1 Buffer overflow and SQL injections examples

Consider a simple buffer overflow, where a GET request is stored on the stack in a buffer that is only 512 bytes long. A normal GET request such as the following will be parsed correctly and the program will return and move on:

```
GET /x.ida?q=N HTTP/1.1
```

A malicious GET request such as the following (where the byte N is repeated over 500 times to overflow the buffer) will cause a return pointer on the stack to be overwritten by the value `0x7801cbd3`:

```
GET /x.ida?q=NNNNNNNN...NNNNN\xd3\xcb\x01\x78
```

Any quantification of information flow for this web server program must be based on some *a priori* probability distribution over the possible code paths (which for a deterministic program is equivalent to a probability distribution over the possible inputs). From the system designer's perspective, this probability distribution is biased towards the inputs they have thought about and tested, namely shorter inputs.

As long as the probability of a longer input is not assumed to be zero, then for either of the inputs above, information flows from the input into the 32-bit return pointer on the stack. This is most obvious for the second input that triggers the buffer overflow, since the return pointer is saturated with 32 bits of input directly. It is less obvious, though, how information flows from the first input into the program counter on the stack, since the program counter is never changed and has its original value. To see why information flows, consider what the unchanged return pointer on the stack implies about the input. It implies that at least one of the first 512 bytes of the input is a newline character or NULL terminator.

Consider also that, if the function that is being called that has the buffer overflow in it is a function that parses GET requests, information had probably already flowed from the input into that return pointer. This is because that function is probably only called if the first three bytes of the input are "`GET`".

So, as the information flows across the boundaries of different levels of abstraction (the operating system, the HTTP protocol, filesystem abstractions that are specific to this web server or operating system, scripting languages, the program counter register in the architecture layer, *etc.*) the information is broken up into pieces that are interpreted differently at the different levels of abstraction. One of these flows of information has a precipitous fracture at the boundary between the C implementation of the protocol and the computer architecture's stack conventions. For one large subset of the full set of possible inputs a small fraction of a bit of information flows across this abstraction boundary. For another large subset of the possible inputs 32 bits flows across the boundary. So, if we picture the amount of information flow as a landscape covering the entire space of possible inputs, the vulnerability is a large precipice. A precipice in this paper means that in the space of possible inputs the information flow associated with an interpretation of the inputs has what can be visualized as a steep cliff. (Note that this precipice has terraces, where slightly more than 8, 16, or 24 bits flows because the buffer is only overflowed by 1, 2, or 3 bytes.)

A good illustration of this type of fracture is SQL injection. Consider the following SQL query that is built using user input for `$name` and `$password`:

```
SELECT * FROM users WHERE name='$name'
    and password='$password'
```

The single quote (`'`) is a special delimiter in the SQL language. The attacker can leave the password blank, and enter this as their username:

```
'; DROP TABLE users -- comment...
```

This causes the SQL server to execute the following SQL code that it receives from the Web server, with the effect of the entire table of users in the database being deleted:

```
SELECT * FROM users WHERE name='';
    DROP TABLE users -- comment...
```

Under normal operation, when there is no attack, the delimiter (`'`) marks clearly where the user input is and how it should be interpreted at different levels of abstraction, but when delimiters appear in the input this causes a precipitous fracture in how the input is interpreted at the boundary between the web scripting language and SQL.

2.2 Generalizability

One important question is, how does the holographic view of vulnerabilities generalize across many different types of vulnerabilities?

Let us consider an abstract model that captures a wide range of different vulnerability types. Suppose Alice and Bob have two separate systems with a network connection, with Eve eavesdropping

in the middle. Alice and Eve have various levels of access to Bob's services, which include HTTP, SSH, and a wide range of other abstractions and protocols. The same is true for Bob and Eve with respect to Alice's services. Alice and Bob each have confidential information, are concerned about the integrity of their system and information, want their system to be available, and use SSL and other cryptography implementations whenever they can.

From Bob's perspective (and therefore the same is true of Alice's perspective by reflection), all interactions of his system are of the form that his system is in some state, he receives some information that may be from Alice or from Eve (he does not know which, but he shares a secret with Alice), and he has information that he assumes neither Alice nor Eve can know or predict (his pool of entropy from, *e.g.*, hard drive timings). Information flows not just over the network but within Bob's system, according to his software, which has vulnerabilities. The information flows according to the rules set forth by the software, and the state is changed based on the flow of information (plus the software can often be overwritten if its integrity is compromised). So there is a dual relationship between the state of the system and the information flow. Any software vulnerability that can be specified in terms of states in this system (which should include a very broad range of vulnerabilities) could also be specified in terms of information flow.

Practically any vulnerability can be seen as a violation of one of: Confidentiality, Integrity, Availability, Authentication, or Non-repudiation [35, 37]. Confidentiality is the most natural to think of in terms of information flow, in this case information is not supposed to flow from high security objects to low security objects. Integrity states that information must not flow from untrusted sources into trusted objects such as return pointers or system files. Note that confidentiality and integrity requirements are often quantitative, *e.g.*, the attacker should not have "undue influence" [41, 5] but a small amount of information must be able to flow for the system to operate. Availability can be viewed as stating that information must flow[1]. Also, with respect to availability, denial of service can be cast as an attack on the integrity of the information the system needs for its operation.

From a systems perspective, authentication is based on what a subject has, knows, or is. The subject carries this information with them and then it is supposed to flow into the system immediately preceding any commands that come into the system that should be interpreted as coming from that subject (*e.g.*, the subject enters their password over SSH and then is given a command shell that persists for the life of the rest of that connection). That information that identified a specific subject should never flow into the system when that specific subject is not present at the terminal that commands will be interpreted from. Take, for example, replay attacks. The information is being copied and then replayed later, so it flows in a way that is not part of the design of the security system.

Another view of authentication is the cryptographic view. In this case the system is using mathematical properties to know what a subject has, knows, or is, rather than physical properties. Cryptography can be seen as being used in this case as a separation mechanism that controls how information flows. Flaws in cryptography are therefore flaws in a separation mechanism that allow information to flow when it should not. Non-repudiation is a special case of authentication where the party performing authentication cannot independently reproduce what the subject being authenticated has, knows, or is, so that they cannot forge authenticated messages.

The purpose of the abstract model above is to illustrate how a broad range of vulnerabilities can be cast as information flow. From Bob's perspective, information flows into his system and changes its state, where some states are safe states and others are states where a security policy has been violated. Since Bob's system can only change states based on the flow of information into the system, any vulnerability that can be described as an unsafe state can also be described in terms of information flow.

A full vulnerability study is outside the scope of this position paper, but what would a vulnerability study based on the holographic view of vulnerabilities look like? Here are three examples of vulnerability classes stated in terms of information flowing across abstraction boundaries:

1. Vulnerabilities where information flows from the security metadata of one object into a security decision about another.
2. Vulnerabilities where information flows from the input to a process into the control flow of the process.
3. Vulnerabilities where information flows from something a subject knows, has, or is into an authentication decision *via* another subject.

Note that all of these information flows occur when there is not a vulnerability; what makes the information flow a vulnerability is when it causes a precipitous fracture in the way the information is interpreted at abstraction boundaries. Without this notion of precipices, there could be ambiguity between vulnerability types.

The example classifications above illustrate two potential advantages of vulnerability studies based on the holographic view: each classification is broad enough to cover many different types of vulnerabilities (*i.e.*, is generalizable), and each vulnerability type should fit into at most one classification (*i.e.*, the classifications are unambiguous). Time-of-check-to-time-of-use (TOCTTOU) and confused deputy vulnerabilities clearly fall into vulnerability type #1. Type #2 covers all control flow hijacking attacks, where the precipitous change in interpretation of the information is illustrated by the Epsilon-Gamma-Pi model [19, 18]. Type #3 covers a wide range of authentication vulnerabilities, from hard-coded passwords to brute force attacks to poorly designed password hash functions. The precipice in these cases is that information from one subject is interpreted and used as an authentication decision for a different subject.

2.3 Mitigation strategies

A common approach taken by the security community in the face of certain vulnerability types are patch-like mitigation solutions. Here we will focus on memory corruption vulnerabilities (*e.g.* heap overflows), and discuss the mitigation strategies: return address protection schemes, ASLR (Address Space Layout Randomization), and $W \otimes X$.

The classical example of return address protection scheme is StackGuard [16]. It protects the stack by inserting additional instructions into the function entry and exit code where the goal is to protect the stack by checking if it is corrupted. Additional function entry code writes a canary value below the saved frame pointer on the stack. Upon exiting, the function exit code checks that the canary was not corrupted, as in the case of a simple stack-based buffer overflow the attacker will need to corrupt the canary to tamper with the return address. Other examples are StackShield [2], ProPolice [1], and SmashGuard [42]. This particular mitigation strategy of protecting the return address on the stack is specific to stack-based buffer overflows that are exploited by overwriting the return pointer, it cannot protect against the many other forms of memory corruption attacks, nor can it protect other data on the stack (such as the structured exception handler function pointer that Code Red overwrote [17]).

[1]This idea was suggested to one of the authors by Stephen Chong.

ASLR randomly arranges the memory positions of key areas such as the heap, stack, and libraries to make the prediction of hard-coded addresses more difficult for the attacker. This idea is implemented in PAX ASLR [44] and is supported by many OSes. The intuition is that in order to exploit a buffer overflow, an attacker needs to predict the location of a return address or know the exact location of a library function. However, ASLR has its own limitations [39, 30].

The $W \otimes X$ defense [53] ensures that no memory location in a process address space can be marked both as writable (W) and executable (X) at the same time. The assumption is that executable code is supposed to be located in the code portion of a process address space and not on the stack, heap, or data areas. Solar Designer's StackPatch [23] modifies the address space of a process to make the stack non-executable. Most OSes now offer such protection. Any defense that prevents the execution of injected code in memory, including $W \otimes X$, can be defeated by code-reuse attacks that exploit existing system code to perform malicious computations. These attacks are based on the technique of return-oriented programming [13, 50, 10], which generalizes return-to-lib-c attacks [39] by chaining short new instructions, called gadgets, that end with a control flow instruction such as `ret`. Several instructions can be combined to form a code-reuse program. Recent research has shown that such exploits can create their own stack without corrupting the system stack and leverage other types of control flow instructions such as `jump` or `call` instead of `ret` [34].

We believe mitigation strategies (and more narrowly focused vulnerability studies) are very important and even essential to increase the level of security of a system. However, we argue that they often target specific exploit techniques or classes of vulnerabilities, and not general classes of vulnerabilities. Another purpose of vulnerability studies should be to classify vulnerabilities in a way that suggests very general mitigations that cover many vulnerability types, even unknown types.

3. TOCTTOU VULNERABILITIES

Time-of-check-to-time-of-use (TOCTTOU) vulnerabilities are one of the oldest and most well-studied types of vulnerabilities. Perhaps the earliest mention of a TOCTTOU vulnerability is the 1976 RISOS study [3], which referred to this type of vulnerability as "asynchronous validation/inadequate serialization". The 1978 Protection Analysis study [28] describes an example of TOCTTOU and has two relevant categories: data consistency over time and serialization. Aslam [7] subdivided all flaw classes into either "synchronization errors" or "condition validation errors."

This kind of vulnerability occurs when privileged processes are provided with some mechanism to check whether a lower-privileged process should be allowed to access an object before the privileged process does so on the lower-privileged process' behalf. If the object or its attribute can change between this check and the actual access that the privileged process makes, attackers can exploit this fact to cause privileged processes to make accesses on their behalf that subvert security.

The classic example of TOCTTOU is the sequence of system calls of `access()` followed by `open()`. The `access()` system call was introduced to UNIX systems as a way for privileged processes (particularly those with an effective user ID of root) to check if the user who owned the process that invoked them (the real user ID) has permissions on a file before the privileged process accesses the file on the real user ID's behalf. For example:

```
if (access("/home/bob/symlink", R_OK | W_OK) != -1)
{
     // Symbolic link can change here
     f = fopen("/home/bob/symlink", "rw");
     ...
}
```

What makes this a vulnerability is the fact that the invoker of the privileged process can cause a race condition where something about the filesystem changes in between the call to `access()` and the call to `open()`. For example, the file `/home/bob/symlink` can be a symbolic link that points to a file the attacker is allowed to access during the `access()` check (*e.g.*, the file `/home/bob/bobsfile.txt` that bob can read and write), but at a critical moment is changed to point to a different file that needs elevated privileges for access (*e.g.*, `/etc/shadow`).

TOCTTOU vulnerabilities are a much broader class of vulnerabilities and are much more of a serious concern for secure system designers than might be suggested by this simple and well-understood example. Not all TOCTTOU vulnerabilities are related to UNIX filesystem atomicity issues. For example, TOCTTOU vulnerabilities in web applications can allow customers to add items to their cart after paying but before shipping, so that they get the items for free [57]. We only use the classical TOCTTOU example because it is easy to understand and demonstrates our points about layers of abstraction and information flow clearly.

How does TOCTTOU fit into our proposed view of vulnerabilities, where a vulnerability is a fracture in interpretation as information flows across layers of abstraction? Consider that the security checks for `/home/bob/bobsfile.txt` (including `stat()`ing each of the dentry's and checking the inode's access control list) get compressed into a return value for the `access()` system call that is returned in a register. This information is interpreted to mean that bob is allowed to access the file referred to by `/home/bob/symlink`. The information crosses the boundary between an OS abstraction (the kernel) and a user-level abstraction in the return value register (architecture layer abstraction). Then a conditional control flow transfer is conditioned on this register, so that through a control flag (potentially) and the program counter the information is now transformed into a decision to open the file pointed to by `/home/bob/symlink`. It is this information flow between the return value and the `open()` system call, which occurs at the level of abstraction of the computer architecture, where the interpretation of the information becomes fractured. To the OS, the value returned was a security property of `/home/bob/bobsfile.txt`. At the architectural level the program counter, *which contains the exact same information,* is implied to be a security property of `/etc/shadow`. The information is the same, but when viewed from different perspectives for the different layers of abstraction that make up the system the interpretation has been fractured.

Here it is important to make a distinction between vulnerability and exploit. When the `access()...open()` sequence of system calls in the example is executed and there is no attacker to change the symlink, it is still implied in the information contained in the access control check that the security information can be about any file that bob can `stat()` and the file that is opened can be any file that the process has privileges to open. What was said about `/home/bob/bobsfile.txt` *vs.* `/etc/shadow` is only precisely true when an exploit occurs and an attacker points the symbolic link at both files for the two system calls `access()` and `open()`, respectively. But, the whole point of a vulnerability study is to generalize the vulnerability, so in terms of the flow and interpretation of information if we consider everything that could have happened

the vulnerable flow and interpretation is implied by the vulnerable system call sequence even when there is not an exploit.

4. LAYERS OF ABSTRACTION

The problem of classifying vulnerabilities is difficult. Bishop and Bailey [8] analyzed many vulnerability taxonomies [3, 28, 32, 7] and showed that they are imperfect because, depending on the layer of abstraction that a vulnerability is being considered in, it can be classified in multiple ways. What makes this classification problem hard is the fact that vulnerabilities cross multiple layers of abstraction and these taxonomies do not take this into account. This cross-layer existence not only makes the problem of classifying vulnerabilities hard but also hinders the development of strong and flexible solutions to remove or mitigate them.

In this paper we argue that even though we may never be able to devise a perfect taxonomy system to study vulnerabilities, the research community should re-think the problem of understanding and mitigating vulnerabilities from a new angle: (i) software systems cannot be guaranteed to be free from vulnerabilities because designers and programmers make mistakes and current verification and testing techniques cannot assure that a significantly complex piece of software meets its specification in the presence of errors or bad inputs; (ii) we can do a much better job of understanding vulnerabilities and defending our computer system assets against them when we accept that their causes and effects cross multiple layers of abstractions (application, compiler, operating system, architecture and network), and (iii) we should devise our defense approaches taking this into account.

Our main point in this section is that even if we can view the problem of vulnerabilities as security information flowing to where it is not supposed to flow, this comes into play exactly because the non-authorized flows cross layers of abstraction. Studies about the nature of vulnerabilities can be fruitful if we consider the role played by the intertwinement of layers of abstraction.

4.1 TOCTTOU crossing layers of abstraction

One of the taxonomies analyzed by Bishop and Bailey [8] was the Protection Analysis study [28]. It contains ten security flaws, where some of them are described below [40]:

1. Improper protection domain initialization and enforcement: vulnerabilities related to the initialization of a system and its programs and the enforcement of the security requirements.
   (a) Improper choice of initial protection domain: vulnerabilities related to an initial incorrect assignment of privileges.
   (b) Improper isolation of implementation detail: vulnerabilities that allow users to bypass a layer of abstraction (*e.g.*, the OS or the architecture) and write directly into protected data structures or memory areas, for instance, I/O memory and CPU registers.
   (c) Improper change: vulnerabilities that allow an unprivileged subject to change the binding of a name or a pointer to a sensitive object so that it can bypass system permissions.
2. Improper validation: vulnerabilities related to improper checking of operands or function parameters.
3. Improper synchronization: vulnerabilities arising when a processes fails to coordinate concurrent activities that might access a shared resource.
   (a) Improper indivisibility: interruption of a sequence of instructions that should execute atomically.
   (b) Improper sequencing: failure to properly order concurrent read and write operations on a shared resource.

From the vulnerable process' perspective TOCTTOU can be classified as 3a (improper indivisibility) as the operation for checking the access permissions and opening the file should have been executed atomically. It can be also viewed as 3b (improper sequencing), as the operations that accessed a specific file object were not properly ordered. From the OS perspective it could be classified as 1c (improper change) as the state of the symbolic link was changed while in use. Further, TOCTTOU can be classified as 1b (improper isolation of implementation detail). For example, Borisov *et al.* [11] described a TOCTTOU attack where the attacker increased their chances to win the race by slowing down file system operations (interfering with the OS).

4.2 Why we cross multiple layers

To understand how this cross-layer existence exacerbates the problem of studying vulnerabilities, let us consider the simplest security model: the gatekeeper. An analogy to the gatekeeper model is a bank teller. Suppose a bank customer needs to use a bank service or access their account. They will walk up to the bank teller's window and give them instructions about what they would like to be done (*e.g.*, withdraw money, deposit, transfer, obtain a cashier's check, *etc.*). The bank teller is a trusted entity that actually carries out these actions on behalf of the customer, while at the same time, ensuring that the bank's policies are being followed. For example, in the case of money withdrawal, the bank policy is to never give out money to a customer before subtracting the same amount from their account. The bank teller ensures that this is done properly when doing transactions on the customers' behalf.

This abstraction represents the gatekeeper concept from an early OS called MULTICS [20, 26]. In MULTICS processes were separated into 64 rings and the lower the ring number some code is executing in, the greater its privileges. The system must be aware of ring crossing at the architecture layer. When code in ring i attempts to transfer control to code in ring j, for the code in ring j to do something on its behalf, a fault occurs and control is given to the OS. The gatekeeper is the software abstraction that handles this fault. Ideally the gatekeeper should be as simple as possible. Simplicity allows for ease of inspection, testing and verification. The design is streamlined and the likelihood of an error is reduced. On the other hand, if the gatekeeper is complex, involving a set of abstractions where many high-level, more complex abstractions are built on top of simpler, lower-level ones in multiple tiers, the possibility of the introduction of an avenue for deceiving the gatekeeper is greatly increased.

Most modern information systems are designed with complex and cross-layered security abstractions. For example, for security reasons a process can only access memory inside its address space. However, the implementation of the abstraction of functions uses the process address space (the stack region) at the user level. This abstraction should be under strict OS control, but processes have access to their address space. Further, sensitive low-level control information (a function return pointer) belonging to the architecture level (value of the program counter register) can be accessed by the OS and processes because it is stored on the stack.

In TOCTTOU, the resource that we would like to be protected by the gatekeeper is the mapping between a symbolic link and a file object. A symbolic link contains a reference to a file object in the form of a path name. Addressing a file object through a path

name is an indirect operation, as the UNIX file system is tree-based and a path name specifies the path taken through the tree to reach a file object [9]. In this case the gatekeeper (or the security policy) does not control which subjects have possession of a link to a file object, neither when nor how they make changes in this reference.

Also security principles, such as the much-referenced Saltzer and Schroeder [47] can conflict with each other and add to the complexity of the gatekeeper. For example, the principle of *least common mechanism* states that the amount of mechanisms that are common to more than one user used to access resources should be minimized. Ideally, access to resources should not be shared by two or more users or should be accomplished through separate channels to prevent unintentional security compromises. One can see the application of this principle in the way UNIX symbolic links are implemented. The accesses to files are not shared by two or more users and can be accomplished through separate channels (different symbolic links referencing the same file object). And, ironically, that is exactly when the unintentional security compromise occurs. The application of this principle is also at odds with the principle of *economy of mechanism* (which states that the design of a secure system or security mechanism should be as simple and as small as possible) and also increases the size, and consequently, the complexity of the gatekeeper.

4.3 Is there a problem when vulnerabilities are put into multiple categories?

Vulnerability classification studies have three main goals [8]: (i) allow systems to be specified, designed and implemented free of vulnerabilities, (ii) allow a computer system to be analyzed to detect vulnerabilities and attempted exploitation, and (iii) allow a system to react against vulnerabilities during its normal operation. When vulnerabilities can be classified in multiple ways or crossing multiple layers of abstraction this implies that the way systems are designed and vulnerabilities are addressed should take this into account, otherwise vulnerabilities cannot be completely prevented or stopped. So, existing vulnerability studies that allow ambiguities in classification can work well for goals (ii) and (iii), but a more general notion of vulnerabilities that identifies root causes is necessary for goal (i).

For example, Dean and Hu [21] provided a probabilistic solution for file system TOCTTOU that relied on decreasing the chances of an attacker to win all races. In this solution, the invocation of the `access()` ... `open()` sequence of system calls are followed by a certain number k of strengthening rounds, which consists of an additional calls to the `access()` followed by `open()`. Analyzing the solution and the problem from the application layer viewpoint, this solution addresses the concurrency issue. Borisov *et al* [11], however, observed that this vulnerability crossed the boundary between the application and OS layers and leveraged this to allow an attacker to easily win the race. They described an attack where an adversary increased their chances to win the race by slowing down file system operations (interfering with the OS). The attacker managed to periodically flush the buffer cache by creating a great number of file objects. In this case, the victim process is very likely suspended after executing the `access()` system call, giving the attacker the opportunity to change the file system. The reason we use TOCTTOU as a running example in this paper is to demonstrate how the holographic view of vulnerabilities can suggest the root cause of any given vulnerability type.

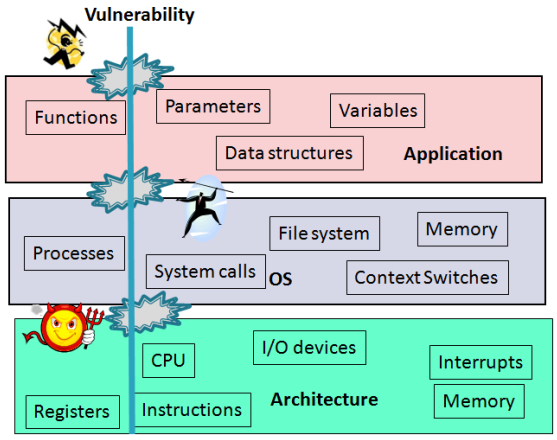

Figure 1: Cross-layered vulnerability.

4.4 A cross-layered approach to understand and mitigate vulnerabilities

Given that vulnerabilities involve more than one layer of abstraction, defending against them should involve some form of collaboration between layers of abstraction to bridge the semantic gap between them.

Most current security solutions operate at one particular level of abstraction. For example, they operate at the application-level as a user level process [43], at the compiler level so that safer binary code is generated [16], at the system level as an OS security extension [4] and also at the architecture level, usually involving a virtual machine layer [25]. This traditional model comes with a cost: the semantic gap problem. There are significant differences between the abstractions or state observed at two distinct layers of abstraction. For example, a user level program deals with abstractions such as functions, parameters, variables and data structures, the operating system works on abstractions such as processes, system calls, kernel data structures, files, and context switches, while at architecture level the abstractions perceived are instructions, CPU, main memory and I/O devices (see Figure 1). The semantic gap hinders the development and widespread deployment of security solutions because these approaches need to inspect and manipulate objects at various levels of abstraction to function correctly.

An approach to mitigate TOCTTOU vulnerabilities should involve an interaction between a user level program and the operating system so that the mapping between the symbolic link object and the file object it references are protected at both levels of abstraction.

5. INFORMATION FLOW

Information flow is fundamental to computer security. A broad range of vulnerabilities can be stated as information flowing from one place to another when it is not supposed to (see Section 2). What we argue in this section is that for TOCTTOU describing the vulnerability in a general way that can help us to address large classes of vulnerabilities in a wholesale fashion entails a deeper understanding of quantitative information flow than the research community currently has. Serious thought must be put into how vulnerability studies can generalize beyond the specific code paths that are attacks.

5.1 TOCTTOU as an information flow problem

For the example TOCTTOU vulnerability and exploit in Section 3, security information about the file `/home/bob/bobsfile.txt` flows into a decision for the privileged process to open the restricted file `/etc/shadow` on an unprivileged user's behalf. This is a vulnerability of type #1 from Section 2.

The information starts on non-volatile storage on the filesystem in the inode for `/home/bob/bobsfile.txt`. This information probably originated largely from bob when he created the file. The inode describes information about the file, including its access controls. An example of an inode as shown by debugfs (with bob assumed to have the user ID and group ID of 1000) is:

```
Inode: 1703953   Type: regular    Mode:  0644   Flags: 0x80000
Generation: 1627390337    Version: 0x00000000:00000001
User:  1000   Group:  1000   Size: 21
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x4f86e402:4f12b4dc -- Thu Apr 12 08:17:38 2012
 atime: 0x4f86e384:28ed64d0 -- Thu Apr 12 08:15:32 2012
 mtime: 0x4f86e384:28ed64d0 -- Thu Apr 12 08:15:32 2012
crtime: 0x4f86e402:4f12b4dc -- Thu Apr 12 08:17:38 2012
Size of extra inode fields: 28
EXTENTS:
(0): 6852607
```

It is imprecise to say that security information is flowing just from the inode, since bob also needed execute permissions to stat all of the dentry's to get to the inode in the directory structure, but for the sake of simplicity just consider the inode itself. The security information in this inode says that bob owns the file, the owner may read and write to it, and others may read it. When the `access()` system call is invoked by the privileged process in the example in Section 3, the semantics of the access system call cause a security check based on the real user ID of the privileged process, which is bob. Since, according to the inode, bob is allowed to read from and write to this file the access system call returns zero to indicate success.

This zero word that gets returned in a register by the `access()` system call is key. It is just a zero word, no different from any other integer stored in a word of memory. It is interpreted to mean that bob can access the `/home/bob/bobsfile.txt` file. But, where is this information headed? If an attacker changes the symbolic link, then through a conditional control flow transfer conditioned on the zero word (the `if` statement in the example) the information contained in this zero word flows into a decision to open a different file (`/etc/shadow` in the example). The access control information for the decision to open `/etc/shadow` using the effective user ID privileges of the setuid process should have come from the inode for `/etc/shadow`, which might look like this:

```
node: 66584725   Type: regular    Mode:  0640   Flags: 0x80000
Generation: 1627370579    Version: 0x00000000:00000001
User:     0   Group:    42   Size: 1196
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x4f79dafd:0c7409cc -- Mon Apr  2 10:59:41 2012
 atime: 0x4f85bc8d:8fbe89c0 -- Wed Apr 11 11:17:01 2012
 mtime: 0x4f79dafd:05309030 -- Mon Apr  2 10:59:41 2012
crtime: 0x4f79dafd:05309030 -- Mon Apr  2 10:59:41 2012
Size of extra inode fields: 28
EXTENTS:
(0): 73005385
```

This inode says that user bob has no permission to read from or write to `/etc/shadow`, since the file is owned by root and other users who are not root and not in root's group are given no permissions. The kernel checks the proper inode for the effective user ID (which is root in the example), but the `open()` system call is only invoked because of an access check for a different file that flows through user space where it has a different meaning than an access check in the kernel actually has.

One interpretation of this vulnerable flow of information is that the zero word becomes outdated because the security information it describes might change. However, UNIX systems have never made guarantees about the atomicity of filesystem operations between two different processes. If two processes have a file open and one overwrites it, this poor synchronization is the fault of the processes and not a semantic bug in UNIX. If a process has a file open with an entry in its file descriptor table and the permissions of the file change, the open file descriptor still allows access but this is not a bug, it is the well-known semantics of UNIX. Thus the lack of atomicity in the `access()...open()` sequence of system calls is a red herring, the real vulnerability lies in the flow of access control information about one file into a decision to open another.

5.2 Why information flow is hard

Information flow is a useful way to think about vulnerabilities, but almost 40 years of research show that tracking all of the meaningful flows of information in a practical system is an intractable problem. In his 1973 Ph.D. thesis Fenton said about information flow systems, "*it must be emphasized that the possibility of proving the correctness of a practical system seems remote. Such a system would need, as a base, a correct address protection system and, in addition, a highly structured operating system*" [24]. The Cambridge CAP [38, 33], to which Fenton was referring, had arguably a richer address protection system and a more structured operating system than modern Pentium-based Windows and Linux systems.

The fundamental problem that prevents us from making progress toward being able to reason about the information flow of a practical system is that everything that *could have happened* carries more information than *what did happen*. In Shannon's information theory [51], a value assigned to a random variable carries no information itself, the information one can learn by measuring the random variable is a function only of what could have happened. What did happen affects the information only by extension of the fact that what did happen came from the set of things that could have happened.

The fact that, for an unknown vulnerability, the attacker's malicious input that causes a precipitous change in information flow is unknown to us is a challenge for any vulnerability study. If we consider all possible inputs (to be conservative and quantitatively measure the information flow in a way that captures all possible vulnerabilities), then the flow of information associated with any vulnerability is very small unless conditioned on the attack input (which we do not know). Consider the following example:

```
if (z == MAGIC)
{
   y = x;
}
else
{
   y = x & 1;
}
```

Here, `x`, `y`, and `z` are 32-bit integers, and are assumed to be distributed equally over the 2^{32} possible integers. `x & 1` is the bitwise AND of `x` and `1`, *i.e.*, the least significant bit of `x`. In general, slightly over a bit flows from `x` to `y`. Conditioned on the common

case that `z != MAGIC`, exactly one bit flows from `x` to `y`. If the attacker can cause `z == MAGIC` to be true (by exploiting a vulnerability), though, then 32 bits of information flow from `x` to `y` conditioned on that code path. This is the nature of most vulnerabilities when you look at them from an information flow perspective. Information usually does not flow at high rates in ways that suggest vulnerabilities in the common case or in general. The attacker makes the specific code path happen that causes a precipitous increase in the flow of information. Attackers make uncommon code paths happen, whereas system designers focus on the common code paths that they know about and are often not aware of the attack code path until the carefully crafted input that causes it is presented to them.

This makes it challenging (but not impossible) to find a vulnerability based on information flow measurements of the system when you do not already know the code path of the exploit. The holographic view of vulnerabilities helps in terms of generalizability and unambiguity in vulnerability classes, but any vulnerability study faces the challenge that vulnerabilities are hidden beneath very specific code paths.

Static analysis methods, such as Denning's lattice model [22], overcome this limitation by analyzing all possible code paths. This can be effective (despite being formally undecidable in the general case), but static analysis methods that can reason about live kernels running on real hardware with interrupts, *etc.*, are typically applied to a limited set of vulnerability types for highly structured operating systems such as seL4 [29].

For a TOCTTOU vulnerability, the tell-tale information flow of access control information about one file leading to a decision to open a different file does occur every time the vulnerable sequence of system calls is executed, even if there is no exploit. The fact that a symbolic link can create concurrent code paths interleaved through the kernel and several userspace processes that would cause the erroneous flow of access control information is implicit in the common case, since information flows based on what *could have happened*. For a general class of vulnerabilities in the holographic view, reasoning about unknown vulnerabilities of that type in a system amounts to a very detailed understanding of the quantitative information flow of the system. Quantitative information flow has seen some recent research, for both static methods [36] and dynamic [5], but much more research into quantitative information flow will be necessary before it can be used to find precipitous fractures in the interpretation of information as it flows across abstraction boundaries, however.

6. RELATED WORK

Bishop and Dilger [9] present a formal language for describing vulnerabilities and a tool to analyze programs for a possible race condition in file accesses. Given a specific vulnerability, they define its signature as the set of minimal attack signatures that exploit it. The tool parses a C program looking for pairs of system calls with the same file name as an argument and builds a dependency graph. From the graph they determine suspicious sequences of system calls. Their assumption was that a race condition is possible when two calls access a file through a file name, which is resolved by indirection and is not directly bound to a file object.

Eraser [49] is a tool for automatic detection of data races in lock-based multithreaded programs which improves Lamport's *happens-before* relation [31]. The *happens-before* order is a partial order on all events of all threads executing concurrently. Within any single thread, events are ordered in the order in which they occurred. Between threads, events are ordered according to the properties of the synchronization objects they access. Eraser uses binary rewriting to monitor every shared-memory reference and verify that all shared memory accesses follow a consistent locking discipline. Eraser monitors all reads and writes as the program executes. For each shared variable v, Eraser maintains the set $C(v)$ of candidate locks for v, or the locks that have protected v for the computation so far. A lock l is in $C(v)$ if in the computation up to that point, every thread that has accessed v was holding l at the moment of the access. When a new variable v is initialized $C(v)$ is initialized to all possible locks and when the variable is accessed $C(v)$ is updated with the intersection between $C(v)$ and the set of locks held by the current thread. The main idea is that if some lock consistently protects v it will remain in $C(v)$ and if $C(v)$ becomes empty a warning is issued indicating that no lock consistently protects v, with detailed information about thread and detection location within the thread. Eraser instruments each load/store in the program and uses a shadow memory for all variables allocated in the heap and global data area to store lock set information.

Although the context of Eraser is not directly security related, Eraser is relevant to the current discussion because it finds race condition bugs without it being necessary for the specific interleaving of threads that causes the bug to occur. If it were possible to define the layers of abstraction and information flow for a large class of TOCTTOU bugs, could we find many such bugs in a system through some kind of analysis that, like Eraser, generalizes beyond specific code paths?

There have been many efforts to detect or prevent race condition attacks [21, 54, 55, 56]. Rouzaud-Cornabas *et al.* [46] is notable because their definition of a TOCTTOU attack is based on information flow. Detecting or preventing attacks is fundamentally different from (and much easier than) detecting vulnerabilities because in the case of the former the attacker has revealed the violating code path to the system. Also, many of the proposed methods for detection or prevention do not have definitions of the vulnerability that are precise enough and are therefore not effective against more advanced attacks [11, 14].

Some defense solutions against TOCTTOU employ the concept of transactions in varying degrees. Tsyrklevich *et al* [55] monitors pseudo transactions during processes execution and suspends one of the processes if a race condition is identified. A pseudo transaction is defined as a sequence of system calls that should be free from vulnerabilities. TxOS [45] provides transactions at the OS level. The concept of transaction is very intuitive and powerful when we think about a strategy to defeat TOCTTOU vulnerabilities. However, transactions only work if one knows the vulnerable sequence of system calls and wrap them in a transaction. One would need to know how to specify and find the vulnerabilities before they could wrap them all in transactions.

Discussions about the theoretical and computational science of exploit techniques and proposals to do explicit parsing and normalization of inputs fit nicely into the discussion of this paper. Bratus *et al.* [12] discuss "weird machines" and the view that the theoretical language aspects of computer science lie at the heart of practical computer security problems, especially exploitable vulnerabilities. Samuel and Erlingsson [48] propose that data confusion vulnerabilities can be effectively prevented by normalizing inputs *via* parsing.

7. WORKSHOP DISCUSSION

The lively discussion at the workshop focused mainly on three topics: (i) the generalizability of the holographic model, (ii) the definition of a layer or boundary of abstraction, and (iii) how we can move forward from this initial model to actual implementation and applications.

Bob Blakley questions whether all vulnerabilities can fit our model. Richard Ford, Anthony Morton, and Ed Talbot discussed the concept of layer of abstraction in the context of our proposed model. Prof. Blakley thinks that layer of abstraction boundaries are not the only type of boundary crossing which can create holographic fractures; crossing domains of semantic interpretation at the same layer of abstraction can also cause a fracture. He suggests a broader class of boundaries than just those between abstraction levels. Prof. Ford agrees and adds that our notion of abstraction layers implies that we need boundaries to get vulnerabilities. Ed Talbot believes layers arise from our own perception of, as Michael Locasto phrases, "boundaries of competence" (*e.g.*, a network, compiler, operating system or hardware competence). According to Ed Talbot we have become highly skilled in horizontal layers but increasingly unaware of vertical problems, which create the opportunity for holographic fractures. Anthony suggests analyzing the problem non-hierarchically where information also flows across boundaries at the same abstraction level and also questions how we define a boundary of abstraction.

Michael Locasto views this as a language problem: we have two components on opposite sides of a boundary where the designer on one side does not understand the language of the designer on the other side. Prof. Blakley suggests that the fracture in interpretation is in fact superimposed states: in terms of the frame problem, we have a situation where the defender has a narrower frame than the attacker, so the defender thinks there is only one interpretation of a particular set of data, but the attacker realizes that there are other interpretations, some of which can be exploited.

Sven Tuerpe and Vaibhav Garg suggest more general applications of the model. For example, can we use fractures as a security mechanism? Can we create fractures in the attacker's structures as a way of interfering with their attacks? Vaibhav Garg suggests applying a similar information flow visualization technique to privacy, so that people can better understand whether there's a holographic fracture in their view of what is happening to their private information.

Finally, Prof. Blakley thinks that maybe we do not need to solve the information flow problem, which might be intractable. One direction is to decrease the language difference on the two sides of the boundary, which can eliminate many hidden paths and make information flows safe even if we do not analyze them.

8. CONCLUSIONS

In this paper we discussed a re-evaluation of vulnerability studies, which have suffered in the past from problems of generalizability and ambiguity. We have argued that vulnerabilities should be seen as fractures in interpretation as information flows across abstraction boundaries. Vulnerabilities occur when information cross boundaries of abstraction and is interpreted differently at each layer. This view explains why current vulnerability studies, which do not take this into account, fail to classify vulnerabilities unambiguously.

Acknowledgments

We would like to thank the NSPW anonymous reviewers, our shepherd, Paul van Oorschot and all the workshop attendeed for valuable feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. #0844880, #0905177, #1017602, and #1149730. Jed Crandall is also supported by the Defense Advanced Research Projects Agency CRASH program under grant #P-1070-113237.

9. REFERENCES

[1] GCC extension for protecting applications from stack-smashing attacks (http://www.research.ibm.com/trl/projects/security/ssp/).

[2] Stack Shield: A Stack Smashing Technique Protection Tool for Linux (http://www.angelfire.com/sk/stackshield/info.html).

[3] R. P. Abbot, J. S. Chin, J. E. Donnelley, W. L. Konigsford, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems. *NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards*, 1976.

[4] N. S. Agency. Security-enhanced linux (http://www.nsa.gov/research/selinux/).

[5] M. I. Al-Saleh and J. R. Crandall. On Information Flow for Intrusion Detection: What if Accurate Full-system Dynamic Information Flow Tracking was Possible? In *Proceedings of the 2010 New Security Paradigms Workshop*, NSPW '10.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[7] T. Aslam. A Taxonomy of Security Faults in the UNIX Operating System, 1995.

[8] M. Bishop and D. Bailey. A Critical Analysis of Vulnerability Taxonomies. *Technical Report CSE-96-11, University of California at Davis*, 1996.

[9] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Technical Report CSE-95-10, University of California at Davis*, 1995.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. *ASIACCS*, March 2011.

[11] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing Races for Fun and Profit: How to Abuse atime. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[12] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation. USENIX *;login*, December 2011.

[13] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. *ACM CCS*, pages 27–38, 2008.

[14] X. Cai, Y. Gui, and R. Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 27–41, Washington, DC, USA, 2009. IEEE Computer Society.

[15] F. Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.

[16] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, pages 63–78, Jan 1998.

[17] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *MICRO*, pages 221–232, December 2004.

[18] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *ACM CCS*, pages 235–248,

November 2005.

[19] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. *DIMVA*, July 2005.

[20] R. C. Daley and J. B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. In *Proceedings of the first ACM Symposium on Operating System Principles*, SOSP '67, pages 12.1–12.8, New York, NY, USA, 1967. ACM.

[21] D. Dean and A. J. Hu. Fixing Races for Fun and Profit: How to Use access(2). In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[22] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.

[23] S. Designer. Stackpatch (http://www.openwall.com/linux).

[24] J. Fenton. Information protection systems. In *Ph.D. Thesis, University of Cambridge*, 1973.

[25] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[26] R. M. Graham. Protection in an Information Processing Utility. In *Communications of the ACM*, volume 11. ACM, 1968.

[27] R. J. Hansen and M. L. Patterson. Guns and Butter: Toward Formal Axioms of Input Validation. Proceedings of the Black Hat Briefings, 2005.

[28] R. B. II and D. Hollingsworth. Protection Analysis Project Final Report. *ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute*, 1978.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[30] T. Kornau. Return-oriented Programming for the ARM Architecture. Master's thesis, Ruhr-Universitat Bochum, 2010.

[31] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[32] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3), 1994.

[33] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[34] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with Return-less Kernels. *EuroSys*, April 2010.

[35] W. V. Maconachy, C. D. Schou, D. Ragsdale, and D. Welch. A Model for Information Assurance: An Integrated Approach. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, 2001.

[36] P. Malacaria. Assessing Security Threats of Looping Constructs. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2007. ACM Press.

[37] J. R. McCumber. Information Systems Security: A Comprehehnsive Model. In *Proceedings of the 14th National Computer Security Conference*. National Institutde of Standards and Technology, 1991.

[38] R. M. Needham and R. D. Walker. The Cambridge CAP Computer and its Protection System. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 1–10, New York, NY, USA, 1977. ACM Press.

[39] Nergal. The Advanced Return-into-lib(c) eEploits: PaX Case Study . *Phrack*, 1(58), December 2001.

[40] P. Neumann. Computer Systems Security Evaluation. *National Computer Conference Proceedings (AFIPS Conference Proceedings)*, pages 1087–1095, 1978.

[41] J. Newsome, S. McCamant, and D. Song. Measuring Channel Capacity to Distinguish undue Influence. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85, New York, NY, USA, 2009. ACM.

[42] H. Odoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. Smashguard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, 55:1271–1285, April 2006.

[43] S. Ortolani, C. Giuffrida, and B. Crispo. KLIMAX: Profiling Memory Writes to Detect Keystroke-Harvesting Malware. *RAID*, 2011.

[44] PaX Project. Address space layout randomization, Mar 2003. `http://pageexec.virtualave.net/docs/aslr.txt`.

[45] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating Systems Transactions. *ACM Symposium on Operating Systems Principles*, 2009.

[46] J. Rouzaud-Cornabas, P. Clemente, and C. Toinard. An Information Flow Approach for Preventing Race Conditions: Dynamic Protection of the Linux OS. In *Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '10, pages 11–16, Washington, DC, USA, 2010. IEEE Computer Society.

[47] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):338–402, 1975.

[48] M. Samuel and U. Erlingsson. Let's Parse to Prevent pwnage (invited position paper). In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, LEET'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[49] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 15(34), 1997.

[50] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). *ACM CCS*, pages 552–561, 2007.

[51] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.

[52] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[53] P. Team. Pax non-executable pages design & implementation (http://pax.grsecurity.net/docs/noexec.txt).

[54] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 13:1–13:18, Berkeley, CA, USA, 2008. USENIX Association.

[55] E. Tsyrklevich and B. Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 17–17, Berkeley, CA, USA, 2003. USENIX Association.

[56] P. Uppuluri, U. Joshi, and A. Ray. Preventing Race Condition Attacks on File-systems. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 346–353, New York, NY, USA, 2005. ACM.

[57] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 465–480. IEEE Computer Society, 2011.

[58] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *30th International conference on Software engineering*, ICSE '08, New York, NY, USA, 2008. ACM.