An Information Flow-Based Taxonomy to Understand the Nature of Software Vulnerabilities

Daniela Oliveira, Jedidiah Crandall, Harry Kalodner, Nicole Morin, Megan Maher, Jesus Navarro, Felix Emiliano

▶ **To cite this version:**

Daniela Oliveira, Jedidiah Crandall, Harry Kalodner, Nicole Morin, Megan Maher, et al.. An Information Flow-Based Taxonomy to Understand the Nature of Software Vulnerabilities. 31st IFIP International Information Security and Privacy Conference (SEC), May 2016, Ghent, Belgium. pp.227-242, 10.1007/978-3-319-33630-5_16 . hal-01369556

HAL Id: hal-01369556
https://inria.hal.science/hal-01369556

Submitted on 21 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

An Information Flow-based Taxonomy to Understand the Nature of Software Vulnerabilities

Daniela Oliveira, Jedidiah Crandall[1], Harry Kalodner[2], Nicole Morin[3], Megan Maher[3], Jesus Navarro[4], and Felix Emiliano[3]
University of Florida University of New Mexico [1] Princeton University[2] Bowdoin College[3] NVIDIA[4]

Abstract. Despite the emphasis on building secure software, the number of vulnerabilities found in our systems is increasing every year, and well-understood vulnerabilities continue to be exploited. A common response to vulnerabilities is patch-based mitigation, which does not completely address the flaw and is often circumvented by an adversary. The problem actually lies in a lack of understanding of the nature of vulnerabilities. Vulnerability taxonomies have been proposed, but their usability is limited because of their ambiguity and complexity. This paper presents a taxonomy that views vulnerabilities as fractures in the interpretation of information as it flows in the system. It also presents a machine learning study validating the taxonomy's unambiguity. A manually labeled set of 641 vulnerabilities trained a classifier that automatically categorized more than 70000 vulnerabilities from three distinct databases with an average success rate of 80%. Important lessons learned are discussed such as (i) approximately 12% of the studied reports provide insufficient information about vulnerabilities, and (ii) the roles of the reporter and developer are not leveraged, especially regarding information about tools used to find vulnerabilities and approaches to address them.

1 Introduction

Despite the security community emphasis on the importance of building secure software, the number of new vulnerabilities found in our systems is increasing with time; The 2014 Symantec Internet Security report announced that 6,787 *new* vulnerabilities occurred in 2013. This represents a 28% increase in the period 2013–2014, compared to a 6% increase in the period 2012–2013 [5]. Further, old and well-studied vulnerabilities, such as buffer overflows and SQL injections, are still repeatedly reported [3].

A common approach to address vulnerabilities is patch-based mitigation targeting specific exploits. This approach may not completely address the vulnerability since it fails to address its essence, and does not generalize well with similar vulnerabilities exploited differently. Take the file-system TOCTTOU vulnerability as an example. Dean and Hu [17] provided a probabilistic solution for filesystem TOCTTOU that relied on decreasing the chances of an attacker to win all races. In their solution, the invocation of the `access()` ... `open()` sequence of system calls is followed by k additional calls to this pair of system calls. From the application layer viewpoint, the solution addresses the concurrency issue because the chances that the attacker will win all rounds are small.

Borisov *et al.* [10], however, observed that this vulnerability crosses the boundary between the application and the operating system layers, and allowed an attacker to win the race by slowing down filesystem operations. This caused the victim process to be likely suspended after a call to `access()`.

A first step towards viewing cyber security as a science is understanding software vulnerabilities scientifically. Weber *et al.* [31] also argue that a good understanding and systematization of vulnerabilities aids the development of static-analysis or model checking tools for automated discovering of security flaws.

Taxonomies decrease the complexity of understanding concepts in a particular field. Taxonomy-based vulnerability studies have been tried since the 70s [7, 18, 21, 8] but they were proved ambiguous by Bishop and Bailey [9], who showed how the same vulnerability was put into multiple categories depending on the layer of abstraction it was being analyzed. The other problem with current taxonomies is their complexity. For example, CWE v1.9 has 668 weaknesses and 1043 pages. Ambiguous and complex taxonomies not only confuse a developer, but also hinder the widespread development of automated diagnosis tools leveraging its categories as points for checks.

This paper introduces a concise taxonomy for understanding the nature of vulnerabilities that views vulnerabilities as fractures in the interpretation of information as it flows in the system. In a seminal paper on computer viruses [15], Cohen said that "*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' behavior, 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. Information, when viewed from the different perspectives for the various levels of abstraction that make up the system (OS, application, compiler, architecture, Web scripting engine, *etc.*), should still basically have the same interpretation. The lack of understanding on the nature of vulnerabilities cause defense solutions to focus on only one perspective (application, compiler, OS, victim process or attacker process) and become just mitigation solutions that are rapidly circumvented by a knowledgeable adversary.

To validate the unambiguity and usefulness of this taxonomy, a machine learning-based [32] study was conducted using a training set of 641 manually classified vulnerabilities from three public databases: SecurityFocus [35], National Vulnerability Database (NVD) [1] and Open Sourced Vulnerability Database (OSVDB) [2]. This manually labeled set was used to train a machine learning classifier built with the Weka suite of machine learning software [32]. More than 70000 vulnerabilities from a ten year period from the three databases were automatically classified with an average success rate of 80%, demonstrating the unambiguity potential of the taxonomy.

Important lessons learned in this study are discussed. First, there are a significant number of poorly reported vulnerabilities (approximately 12% of the vulnerabilities in the manually classified set), with descriptions containing insufficient or ambiguous information. This type of report pollutes the databases and makes it hard to address vulnerabilities scientifically, and disseminate relevant information to the security community. Second, the roles of the reporter and the developer are not leveraged and important information has not been added to reports, such as tools used to find vulnerabilities and

approaches taken to address them. Finally, the lack of standards on vulnerability reports and across databases adds complexity to the goal of addressing vulnerabilities scientifically, as they are viewed as dissimilar, independent and unique objects. The paper also discusses the application of such taxonomy in the context of automated diagnosis tools to assist the developer.

This paper's contributions are as follows:

1. A concise taxonomy for understanding the nature of vulnerabilities based on information-flow that can be easily generalized and understood is proposed.
2. The taxonomy's categories and their information-flow nature are discussed against notorious vulnerabilities, such as buffer overflows, SQL injection, XSS, CSRF, TOCTTOU, side-channels, DoS, *etc.*.
3. A large scale machine learning study validating the taxonomy's unambiguity is presented. In this study a manually labeled set of 641 vulnerabilities trained a classifier that automatically categorized more than 70000 vulnerabilities from three distinct databases with an average success rate of 80%.
4. Important lessons learned are discussed such as (i) approximately 12% of the studied reports provide insufficient information about vulnerabilities, and (ii) the roles of the reporter and developer are not leveraged, especially regarding information about tools used to find vulnerabilities and approaches to address them.
5. A discussion of the application of this taxonomy in automated diagnosis tools is provided.

The rest of the paper is organized as follows. Section 2 presents the proposed taxonomy and discusses notorious vulnerabilities from the perspective of information flow. Section 3 presents the machine learning study conducted to evaluate the taxonomy. Section 4 discusses related work and section 5 concludes the paper.

2 The Taxonomy

This paper introduces a new vulnerability taxonomy based on information flow. The goal was to produce an unambiguous taxonomy that can be leveraged to address software vulnerabilities scientifically. Vulnerabilities are viewed as fractures in the interpretation of information as it flows in the system. Table 1 details with examples the proposed taxonomy and its categories. The following sections describe each one of these categories with some examples and how they can be viewed in terms of information flow.

Please notice that there is no *design flaw* category because this study understands that all vulnerabilities are ultimately caused by design flaws. Vulnerabilities are weaknesses in the design and/or implementation of a piece of software that allow an adversary to violate the system security policies regarding the three computer security pillars: confidentiality, integrity and availability.

2.1 Control-flow hijacking

These vulnerabilities allow an attacker to craft an exploit that communicates with a process in a malicious way, causing the adversary to hijack the process' control-flow.

Category	Description	Examples
Control-flow hijacking	Vulnerabilities where information flows from the input to a process into the control flow of the process causing its execution to be hijacked.	Buffer overflows, memory corruption, SQL injection, cross-site scripts
Process confusion	Vulnerabilities where information flows from the security metadata of one object into a security decision about another.	TOCTTOU, confused deputy, cross-site request forgery (CSRF)
Side-channels	Vulnerabilities where information flows from physical or side-effects of the operation or communication channels of the system into an illegitimate authentication decision or information disclosure.	Physical: timing/power and electromagnetic attacks. Communications/operation: man-in-the-middle, replay, /proc filesystem attacks
Exhaustion	Vulnerabilities where a significant amount of information flows into a process causing unavailability (exhaustion of resources) or an illegitimate authentication decision (exhaustion of input space).	Resources: DoS, TCP SYN flood, ICMP flood. Input space: password cracking and dictionary attacks
Adversarial accessibility	Vulnerabilities where information is allowed to flow to the attacker's process causing a breach of confidentiality, illegitimate authentication or interference with system functionality.	Assignment of weak permissions to system objects, access control errors, and non-control-data attacks [14]

Table 1: Taxonomy categories.

There are several vulnerabilities that fall into this category: all types of buffer overflows [20] (stack, heap, data, dtors, global offset table, *setjmp* and *longjmp*, double-frees, C++ table of virtual pointers, *etc.*), format string, SQL injection [28] and cross-site scripts (XSS) [30]. Code-reuse attacks [26] are considered a capability of an attacker after leveraging a stack-based buffer overflow and not a vulnerability in itself.

In a general memory corruption attack an adversary provides a victim process with a set of bytes as input, where part of these bytes will overwrite some control information with data of the attacker's choice (usually the address of a malicious instruction). This control information contains data that will eventually be loaded into the EIP register, which contains the address of the next instruction to be executed by the CPU at the architecture level.

For these cases, the fracture in the interpretation of information occurs when user input crosses boundaries of abstractions. User input is able to influence the OS, which manages the process address space and the control memory region being abused. User input also influences the architecture layer as it is directly written into the EIP register. For buffer overflows on the heap, data, and *dtors* areas, an attacker overwrites a data structure holding a function pointer with a malicious address. The effect is the same in all cases: the function will be eventually called, and its address will be loaded into the EIP register.

In a SQL injection [28] user input is directly combined with a SQL command written by an application developer, and this allows an attacker to break out of the data context when she supplies input as a combination of data, control characters and her own code. This malicious combination causes a misinterpretation of data input as it is provided by the web scripting engine. The scripting engine, which processes user input,

misinterprets it as data that should be concatenated with a legitimate command created by the application developer. The SQL query interpreter then parses the input provided by the scripting engine as SQL code that should be parsed and executed. The misinterpretation between the web scripting engine and the SQL query interpreter causes the vulnerability.

2.2 Process confusion

This type of vulnerability allows an attacker to confuse a process at a higher layer of abstraction where this process is usually acting as a deputy, performing some task on behalf of another lower privileged process. A fracture in the interpretation of information allows the security metadata of one object to be transferred into a security decision about another object. A classic example is TOCTTOU, one of the oldest and most well-studied types of vulnerability [23]. It 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 either 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 `access()` followed by `open()`:

```
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.*, file `/home/bob/bob.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`).

Consider that the security checks for `/home/bob/bob.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 stored in register EAX. 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 into the EAX register, which contains the return value (architecture layer abstraction). Then a control flow transfer conditioned on the EAX register is now transformed into a decision to open the file pointed to by `/home/bob/symlink`. The interpretation of information becomes fractured in this information flow between

the return value and the `open()` system call, which occurs at the architecture layer. To the OS, the value returned in register EAX was a security property of `/home/bob/bob.txt`. At the architectural level the value of the program counter (register EIP), *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.

TOCTTOU is a much broader class of vulnerabilities and no all cases are related to UNIX filesystem atomicity issues [29].

2.3 Side-channels

This type of vulnerability allows an attacker to learn sensitive information about a system such as cryptographic keys, sites visited by a user, or even the options selected by the user when interacting with web applications by leveraging physical or side-effects of the system execution or communications.

Examples of such vulnerability are found in systems where the execution of certain branches is dependent on input data, causing the program to take varying amounts of time to execute. Thus, an attacker can gain information about the system by analyzing the execution time of algorithms [12]. Other physical effects of the system can be analyzed, such as hardware electromagnetic radiation, power consumption [27] and sound [34]. An attacker can also exploit weaknesses in the communication channels of a process to breach confidentiality [13, 33, 19].

As example, first consider a timing attack (*Physical side-channel*) where an adversary attempts to break a cryptosystem by analyzing the time a cryptographic algorithm takes to execute [12]. The cryptographic algorithm itself does not reveal cryptographic keys, but the leaking of timing information is a side-effect of its execution. This information flows from the server machine to the client machine and is interpreted in the client (the attacker's machine) as tokens of meaningful information. The combination of these tokens of information over several queries allows the attacker to succeed by making correlations among the input, the time to receive an answer, and the key value.

Another example is a Man-in-the-middle (MiM) vulnerability (*Communications / Operation*), which is a form of *eavesdropping* where the communication between two parties, Alice and Bob, is monitored by an unauthorized party, Eve. The eavesdropping characteristic of MiM vulnerabilities implies that authentication information is *leaked* through a channel not anticipated by the system designer (usually the network). In the classic example, Alice asks for Bob's public key, which is sent by Bob through the communication channel. Eve is able to eavesdrop the channel and intercepts Bob's response. Eve sends a message to Alice claiming to be Bob and passing Eve's public key. Eve then fabricates a bogus message to Bob claiming to be Alice and encrypts the message with Bob's public key. In this attack information flows from the communication channel between Alice's and Bob's processes into an illegitimate authentication decision established by Eve.

2.4 Exhaustion

This type of vulnerability allows an adversary to compromise the availability or confidentiality of a system by artificially increasing the amount of information the system needs to handle. This augmented information flow can leave the system unable to operate normally (attack on availability) or can allow an attacker to illegitimately authenticate herself into the system (attack on confidentiality). The *Exhaustion* category was subdivided into two subcategories (exhaustion of resources and exhaustion of input space) due to their differences in nature and also because they target different security pillars, respectively availability and confidentiality. They both belong to the same broader category because they leverage an artificial increase in the amount of information flowing into the system.

Exhaustion of resources vulnerabilities allow an attacker to cause a steep consumption of a system's computational resources, such as CPU power, memory, network bandwidth or disk space. A classic example is the standard DoS attack: an attacker saturates a target machine with communication requests so that the machine is left short of resources to serve legitimate requests. The victim server process does not handle the uncommon case (exploited by attackers) of a steep increase in the amount of information it has to handle.

Exhaustion of input space vulnerabilities are leveraged to allow an adversary to illegitimately authenticate herself into the system by exploiting a great portion of a vulnerable process authentication input space. For example, in a password cracking attack an adversary repeatedly attempts password strings in the hope that one of them will allow her to authenticate herself into the system. A system will be vulnerable to this type of attack depending on the strength of the password. A secure system can tolerate a steep increase in authentication information flowing into it (password guesses) without its confidentiality being compromised, or guard itself against an exhaustion attack, by for example, locking the system after a few failed attempts.

2.5 Adversarial Accessibility

These vulnerabilities occur when weaknesses in the system design and implementation allow information to flow to an adversary or her process when it should not, as per the system security policies. A classic example is when weak permissions are assigned to system objects, allowing an adversary access to sensitive information or abstractions. This illegitimate information flow to the attacker can also result in authentication breaches. For instance, a vulnerable access control mechanism that does not perform all necessary checks can allow an attacker to authenticate herself in the system and access its resources.

3 Evaluation

The goal of this study was to evaluate how faithfully the categories reflect real vulnerabilities and to assess the taxonomy's potential for classifying vulnerabilities unambiguously. This analysis leveraged three well-known public vulnerability databases:

Category	Database/ID	Description (Abridged)
Control-flow hijacking	SF 54982	*"glibc is prone to multiple stack-based buffer-overflow vulnerabilities because it fails to perform boundary checks on user-supplied data".*
Process confusion	NVD 2013-2709	*"Cross-site request forgery vulnerability in the FourSquare Checkins plugin allows remote attackers to hijack the authentication of arbitrary users".*
Side-channels	OSVDB 94062	*"RC4 algorithm has a cryptographic flaw .. the first byte output by the PRG ... correlating to bytes of the key ... allows attacker to collect keystreams to facilitate an attack".*
Exhaustion	NVD 1999-1074	*"Webmin does not restrict the number of invalid passwords that are entered for a valid username, ... allow remote attackers to gain privileges via brute force password cracking.*
Adversarial accessibility	NVD 2013-0947	*"EMC RSA Authentication Manager allows local users to discover cleartext operating-system passwords ... by reading a log file or configuration file."*
No information	SF 55977	*"Oracle Outside In Technology is prone to a local security vulnerability. The 'Outside In Filters' sub component is affected. Oracle Outside In Technology is vulnerable."*
Ambiguous	SF 39710	*JBoss is prone to multiple vulnerabilities, including an information-disclosure issue and multiple authentication-bypass issues. An attacker can exploit these issues to bypass certain security restrictions to obtain sensitive information..."*

Table 2: Examples of manually classified vulnerabilities.

SecurityFocus (SF) [4], National Vulnerability Database (NVD) [1], and Open Source Vulnerability Database (OSVDB) [2].

The study employed machine learning to classify a large number of vulnerabilities according to the proposed taxonomy. In this analysis we used the Weka data mining software [32]. The study started with the *manual* classification, according to the proposed taxonomy, of 728 vulnerabilities from SecurityFocus (202 vulnerabilities), NVD (280 vulnerabilities), and OSVDB (246 vulnerabilities) databases. This manual classification was done independently by four of the authors, with an inter-rater agreement of approximately 0.70 (see Table 2). A vulnerability report contains the following attributes (names vary per database): ID, title, description, class, affected software and version, reporter, exploit and solution. For purposes of classification, the most important attributes in a vulnerability report are the title and the description. The *class* attribute was observed to be highly ambiguous; SecurityFocus, for instance, classifies highly distinct vulnerabilities as *Design error*. The manual classification selected vulnerabilities in descending chronological order, starting with the most recent vulnerabilities in the respective databases. As some categories were under-represented in the most recent set of reported vulnerabilities and the goal was to build a large and well-represented training set, the authors manually searched for reports fitting under-represented categories in the past. This process showed that the taxonomy was easily applied, even though some

questions were raised about vulnerabilities with poor or ambiguous descriptions. Table 3 shows a summary of the manual classification.

Database	Control-flow hijacking	Process confusion	Side channels	Exhaustion	Adversarial accessibility	No info	Ambiguous
SF (202)	60 (30 %)	32 (16%)	27 (13%)	34 (17%)	18 (9%)	17 (8%)	14 (7%)
NVD (280)	149 (53 %)	8 (3%)	24 (8%)	35 (12%)	30 (11%)	11 (4%)	23 (8%)
OSVD (246)	150 (61%)	9 (4%)	26 (10%)	32 (13%)	15 (6%)	8 (3%)	3 (1%)
Total (728)	359 (49%)	49 (7%)	77 (10%)	101 (14%)	63 (9%)	36 (5%)	51 (7%)

Table 3: Manual classification of vulnerabilities.

Approximately 12% of the most recent vulnerability reports contain insufficient or ambiguous information to reason about the corresponding security flaw. For example, the SecurityFocus vulnerability report with BID 55977 only reveals that a certain software is vulnerable. To avoid polluting the training set and confusing the machine learning classifier, all vulnerabilities with insufficient or ambiguous descriptions (87 total) were filtered out of the manually labeled set.

The study proceeded with the automated extraction of all vulnerability reports from NVD, OSVDB and SecurityFocus for the periods of 2013-2012, 2009-2008, and 2004-2003. The goal was to classify vulnerabilities from three distinct periods over the last decade and identify trends and patterns. A total of 70919 vulnerabilities were extracted (37030 from OSVDB, 23155 from NVD and 10506 from Security Focus) forming the testing set to be categorized by the machine learning classifier. We used the Naïve Bayes algorithm as it is popular for text classification.

All the reports collected for the training and testing set were pre-processed by a parser that converted them into the Weka's ARFF format [32]. The parser used the Weka's String to Word vector filter [32], which turned each word in the title or description into an attribute, and checked whether or not it was present. The filter removed stopwords and established a threshold on the number of words kept per machine learning sample.

Table 4 summarizes the results obtained for the automated classification of vulnerabilities for the three databases studied. *Control-flow hijacking* vulnerabilities make more than 50% of all reported vulnerabilities in all databases, followed by *Adversarial accessibility* (19%), *Exhaustion* (16%), *Side-channels* (3%) and *Process confusion* (2%). This trend was consistent in all databases and did not change much over the last decade.

The standard method of stratified tenfold cross validation [32] was used to predict the success rate of the classifier, which obtained, respectively, success rates of 84.6%, 73.1%, and 82% for the OSVDB, NVD, and SecurityFocus databases. The authors believe that two reasons prevented the classifiers from obtaining higher success rates: (i) the non-negligible number of reports with insufficient information about the vulnerability; approximately 12% for the most recent vulnerabilities appearing in the training set for all three databases, and (ii) DoS vulnerabilities, which depending on how they are exploited can be classified as *Exhaustion* or *Control-flow hijacking*. For example,

Period	Total	Control-flow hijacking	Process confusion	Side-channels	Exhaustion	Adversarial accessibility
OSVDB						
2013-12	14270	9261 (64.8%)	555 (3.8%)	440 (3%)	1521 (10.6%)	2493 (17.4%)
2009-08	16945	10770 (63.5%)	66 (0.4%)	126 (0.7%)	3099 (18.2%)	2884 (17%)
2004-03	5815	2990 (51.4%)	0	21 (0.3%)	1318 (22.6%)	1486 (25.5%)
All	37030	23021 (62.1%)	621 (1.6%)	587 (1.5%)	5938 (16%)	6863 (18.5%)
NVD						
2013-12	7822	4062 (51.9%)	239 (3%)	321 (4.1%)	1141 (14.5%)	2059 (26.3%)
2009-08	11361	7021 (61.7%)	207 (1.8%)	310 (2.7%)	1388 (12.2%)	2435 (21.4%)
2004-03	3972	1958 (49.2%)	57 (1.4%)	132 (3.3%)	690 (17.3%)	1135 (28.5%)
All	23155	13041 (56.3.1%)	503 (2.1%)	763 (3.2%)	3219 (13.9%)	5629 (24.3%)
SecurityFocus						
2013-12	2071	1057 (51%)	122 (5.8%)	60 (2.8%)	335 (16.1%)	497 (23.9%)
2009-08	5788	4216 (72.8%)	172 (2.9%)	168 (2.9%)	661 (11.4%)	571 (9.8%)
2004-03	2647	710 (26.8%)	1264 (47.7%)	512 (19.3%)	1264 (47.7%)	139 (5.2%)
All	10506	5983 (56.9%)	316 (3%)	750 (7.1%)	2260 (21.5%)	1207 (11.4%)
All databases consolidated						
2013-12	24163	14380 (59.5%)	916 (3.7%)	820 (3.3%)	2997 (12.4%)	5049 (20.8%)
2009-08	34094	22007 (64.5%)	445 (1.3%)	604 (1.7%)	5148 (15%)	5890 (17.2%)
2004-03	12434	5658 (45.5%)	1321 (10.6%)	665 (5.3%)	3272 (26.3%)	2790 (22.4%)
All	70691	42045 (59.4%)	1440 (2%)	2100 (2.9%)	11417 (16.1%)	13699 (19.3%)

Table 4: Automated classification of vulnerabilities.

an attack that works by sending a very large number of requests to a server, so as it does not have sufficient resources to serve legitimate requests exploits an *Exhaustion* vulnerability. On the other hand, a buffer overflow that crashes the application (still changing the control-flow according to the attacker's choice) is usually named a DoS attack in vulnerability reports, even though the root cause of the vulnerability does not involve exhaustion of resources. Table 5 shows examples of vulnerabilities automatically categorized by the classifier.

3.1 Discussion

Approximately 12% of all examined reports do not provide sufficient information to understand the corresponding vulnerabilities. These descriptions specify the capabilities of attackers after the vulnerability is exploited, or just mention that an unspecified vulnerability exists.

Also, important information on the process of finding vulnerabilities is usually not provided: reporter contact information, tools used to discover vulnerabilities, whether the vulnerability was discovered through normal software usage or careful inspection, exploit examples and steps to reproduce the vulnerability. Certain reports provide URLs for exploits or steps to reproduce the flaw, but many of these links are invalid as if this information were ephemeral. This information should be permanently recorded; it is

Category	Database/ID	Description (Abridged)
Control-flow hijacking	NVD 2003-0375	*"XSS vulnerability in member.php of XMBforum XMB allows remote attackers to insert arbitrary HTML and web script via the "member" parameter."*
Process confusion	OSVDB 94899	*"DirectAdmin Backup System contains a flaw as an unspecified email account function creates temporary files insecurely. It is possible for attacker to use a symlink attack against an unspecified file to gain elevated privileges".*
Side-channels	OSVDB 95626	*"WhatsApp Messenger contains a flaw triggered when attacker intercepts a payment request via a MiM attack ... allow the attacker to redirect user to arbitrary web page".*
Exhaustion	SF 58500	*"IBM Integrator is prone to a DoS vulnerability. Remote attackers can exploit this issue to cause an application to consume excessive amounts of memory and CPU time, resulting in a DoS condition".*
Adversarial accessibility	NVD 2013-3431	*Cisco Video Surveillance Manager does not require authentication for access to VSMC monitoring pages, allows remote attackers to obtain sensitive configuration information.*

Table 5: Examples of vulnerabilities automatically categorized by the classifier.

invaluable to educate developers during the software development cycle and help the security community build a body of knowledge about the nature of vulnerabilities.

The lack of this important information in vulnerability reports shows that the roles played by reporters and developers are undermined. Reports discussing strategies for finding vulnerabilities could help developers designing more secure software. Further, it would be invaluable to the security community and other developers information on how the vulnerability was addressed. For example, was the vulnerability caused by a weakness on a particular API ? Did the developer use a particular tool or strategy to address the vulnerability?

A lack of standardization among vulnerability reports across databases was also observed. This makes it very difficult to understand actual trends and statistics about vulnerabilities; they are viewed as one of a kind and not addressed together according to their similarities. Finally, there is no guarantee that a vulnerability is reported in a public database only after the vendor had been informed about the issue. A responsible reporter should always report the vulnerability first with the vendor or developer and allow them a reasonable amount of time (*e.g.*, 30 days) to address the issue before making it public in a database.

4 Related Work

The first efforts towards understanding software vulnerabilities happened in the 70s through the RISOS Project [7] and the Protection Analysis study [18]. Landwehr *et al.* [21] proposed a taxonomy based on three dimensions: genesis, time, and location, and classified vulnerabilities as either intentional (malicious and non-malicious) or inadvertent. Aslam [8] introduced a taxonomy targeting the organization of vulnerabilities into a database and also the development of static-analysis tools. Bishop and Bailey [9] analyzed these vulnerability taxonomies and concluded that they were imperfect because,

depending on the layer of abstraction that a vulnerability was being considered in, it could be classified in multiple ways.

Lindqvist and Jonsson [22] presented a classification of vulnerabilities with respect to the intrusion techniques and results. The taxonomy on intrusion techniques has three global categories (Bypassing Intended Controls and Active and Passive Misuse of Resources), which are subdivided into nine subcategories. The taxonomy on intrusion results has three broader categories (Exposure, Denial of Service and Erroneous Output), which are subdivided into two levels of subcategories.

More recently the Common Weakness Enumeration (CWE) [6] was introduced as a dictionary of weaknesses maintained by the MITRE Corporation to facilitate the use of tools that can address vulnerabilities in software. The Open Web Application Security Project (OWASP) was also created to raise awareness about application security by identifying some of the most critical risks facing organizations. Even though these projects do not define themselves as taxonomies, their classification is ambiguous. For example, CWE-119 and CWE-120 are two separate weaknesses that address buffer overflows. Also, OWASP classifies *injection* and XSS as different categories, even though XSS concerns malicious code being injected into a web server.

There are also discussions about the theoretical and computational science of exploit techniques and proposals to do explicit parsing and normalization of inputs [11, 25, 16, 24]. Bratus *et al.* [11] 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 [25] propose input normalization via parsing as an effective way to prevent vulnerabilities that allow attackers to break out of data contexts. Crandall and Oliveira [16] discussed in a position paper the information-flow nature of software vulnerabilities.

In this work vulnerabilities are viewed as fractures in the interpretation of information as it flows in the system. It is not attempted to pinpoint a location for a vulnerability because they can manifest in several locations or semantic boundaries. Further, the primary goal of our taxonomy is to address ambiguity, which makes it difficult to reason about vulnerabilities effectively.

5 Conclusions

This paper presented a new vulnerability taxonomy that views vulnerabilities as fractures in the interpretation of information as it flows in the system. Notorious vulnerabilities are discussed in terms of the taxonomy's categories. A machine learning study evaluating the taxonomy is presented. Almost 71000 vulnerabilities were automated classified with an average success rate of 80%. The results showed the taxonomy's potential for unambiguous understanding of vulnerabilities. Lessons learned were discussed: (i) control-flow hijacking vulnerabilities represent more than 50% of all vulnerabilities reported, a trend that was not changed over the last decade, (ii) approximately 12% of recent vulnerabilities reports have insufficient information about the security flaw, (iii) the lack of standards in reporting makes it difficult to address vulnerabilities scientifically. This work will hopefully shed light on how the security community

should approach vulnerabilities and how to best develop automatic diagnostic tools that find vulnerabilities automatically across layers of abstraction.

References

1. National Vulnerability Database (http://nvd.nist.gov/home.cfm).
2. Open Source Vulnerability Database (http://www.osvdb.org/).
3. Security Focus Vulnerability Notes, bugtraq id 66483 - (`http://www.securityfocus.com/bid/66483`).
4. SecurityFocus (http://www.securityfocus.com/).
5. Symantec - Internet Security Threat Report (`http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf`).
6. The Common Weakness Enumeration (CWE) `http://nvd.nist.gov/cwe.cfm`.
7. 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.
8. T. Aslam. A Taxonomy of Security Faults in the UNIX Operating System, 1995.
9. M. Bishop and D. Bailey. A Critical Analysis of Vulnerability Taxonomies. *Technical Report CSE-96-11, University of California at Davis*, 1996.
10. 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.
11. 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.
12. D. Brumley and D. Boneh. Remote timing attacks are practical. *USENIX Security*, 2003.
13. S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. *IEEE Symposium on Security and Privacy*, 2010.
14. S. Chen, J. Xu, and E. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security*, 2005.
15. F. Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.
16. J. Crandall and D. Oliveira. Holographic Vulnerability Studies: Vulnerabilities as Fractures in Terpretation as Information Flows Across Abstraction Boundaries. *New Security Paradigms Workshop (NSPW)*, 2012.
17. 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.
18. R. B. II and D. Hollingsworth. Protection Analysis Project Final Report. *ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute*, 1978.
19. S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints). *IEEE Symposium on Security and Privacy*, 2012.
20. Kyung-Suk and S. J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. *Software - Practice and Experience*, 33:423–460, April 2002.
21. 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.
22. U. Lindqvist and E. Jonsson. How to Systematically Classify Computer Security Intrusions. *IEEE Symposium on Security and Privacy*, 1997.

23. W. S. McPhee. Operating System Integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.
24. W. Pieters and L. Consoli. Vulnerabilities and responsibilities: dealing with monsters in computer security. *Journal of information, communication and ethics in society*, 7(4):243–257, 2009.
25. 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.
26. 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.
27. L. Spadavecchia. A network-based asynchronous architecture for cryptographic devices. *Edinburgh Research Archive*, 2005.
28. 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.
29. 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.
30. 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.
31. S. Weber, P. A. Karger, and A. Paradkar. A Software Flaw Taxonomy: Aiming Tools at Security. *Software Engineering for Secure Systems (SESS)*, 2005.
32. I. W. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.
33. K. Zhang and X. Wang. Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems. *USENIX Security*, 2009.
34. L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Conference on Computer and Communications Security (CCS)*, 2005.
35. Security Focus Vulnerability Notes, (http://www.securityfocus.com), bid == Bugtraq ID.