Classes of Vulnerabilities and Attacks

Pascal Meunier

Keywords

Vulnerability; Attack; Ontology; Classification; Taxonomy; Security; Secure Programming; Weaknesses; Enumeration

Abstract

In the first part of this chapter, popular vulnerability and attack types used in books, vulnerability disclosures and databases are reviewed. They are discussed in the context of what makes them useful, and how they fail to meet scientific criteria, without going into the exploit details. Practical efforts such as the various MITRE enumerations and lists of common security problems by other organizations are also reviewed. The second part discusses attempts at scientific classifications, with the reasoning leading to the current state of the art. The third part discusses ongoing and future research with a focus on ontologies.

For the purposes of this chapter, vulnerabilities are discussed regardless of whether they are actually exploitable. In this context they are functionally equivalent to security flaws (see later the classification by exploitability). A better understanding of vulnerabilities and attacks can be achieved by grouping them based on common properties and similarities. Many groups and "types" are commonly discussed in computer security texts and secure programming materials. These popular classifications usually capture a defect or a weak technology that enables attacks or present an appealing, succinct and useful point of view for discussing vulnerabilities. It is common to refer to the set of vulnerabilities that enable attack scenario X as "X vulnerabilities", e.g., "cross-site scripting vulnerabilities". Sometimes the name of a technology is used instead of the name of an attack, e.g., "format string vulnerabilities". However, the popular vulnerability types suffer from many defects, such as ambiguities and overlapping classifications. A single vulnerability may belong to several popular types simultaneously or at different times, when the analysis is performed from a different point of view or from a different level of abstraction. A systematic grouping can achieve the status of a scientific classification if it meets rigorous criteria, such as reproducibility, objectivity, and lack of ambiguity. First, popular classifications are reviewed and their usefulness discussed. Then, scientific taxonomies and other systematic efforts are discussed. Finally, the advantages of using ontologies instead are presented.

Popular Vulnerability Classifications

Popular vulnerability classifications fail to meet scientific criteria; yet they can be useful and powerful descriptions of what went wrong. For example, in the book "19 deadly sins", Howard et al. use a mix of several different popular classifications [1]. As a result, the theme of each section is easy to understand and can be effectively and succinctly described. However, it is possible to find

examples or contrive vulnerabilities that could fit in more than one section at once, and others that would fit in none[1]. It has also been argued that taxonomies should be judged on whether they are useful, instead of whether they are scientifically correct [2]. In this chapter, classifications aiming first for usefulness are called "popular classifications" to distinguish them from attempts to meet the rigorous scientific criteria for taxonomies established across disciplines. Popular classifications may have different goals from scientific taxonomies, such as educational, conceptual, tool use, etc…

Classification by Software Development LifeCycle (SDLC) Phase

Taxonomies of this kind attempt to categorize vulnerabilities according to when they were introduced in the software lifecycle. Classically, 6 phases are recognized: feasibility study, requirements definition, design, implementation, integration and testing, and operations and maintenance. At a basic level, this classification groups the above phases into 3 groups: design (first 3 phases), implementation (phases 4 and 5), and operation and maintenance. It was suggested that the timing of reviews could be decided based on the phase in which a vulnerability type could be introduced [3].

The design and implementation distinction is particularly appealing to computer scientists who want to argue the correctness of an algorithm, protocol or model in theory, separately from an implementation that may be subject to unfavorable limitations or mistakes resulting in vulnerabilities. For example, a vulnerability may be due to a bad algorithm being chosen during design phases. Another may be due to a bad implementation of a correctly chosen algorithm, and therefore the vulnerability was introduced at some point during the implementation phases of the program.

It was also suggested to classify vulnerabilities into design, implementation and configuration vulnerabilities [4]. However, "configuration" vulnerabilities is a narrower category than all the vulnerabilities introduced during operation and maintenance. This difference could be resolved if maintenance is considered as re-occuring design and implementation phases, and if emergent vulnerabilities are considered to be design vulnerabilities at the level of the system.

Even though operation and maintenance are often seen as a single phase from a timeline point of view, they differ in how they can introduce vulnerabilities. Vulnerabilities introduced during operation are usually configuration issues, although some also emerge due to interactions with an unexpected or changed environment, including other installed software and operating system version, kernel modules and libraries, or unexpected or virtual hardware. Vulnerabilities can be introduced during maintenance to fix bugs or modify functionality. The difference is that maintenance changes the code whereas operation does not.

[1] This is not a criticism of the book, which meets its goal of providing "a simple list of those security issues that are most important" [1]. It was not intended to provide complete, scientific coverage.

So, 5 classes have been proposed: analysis, design, implementation, deployment, and maintenance [5].

The difficulty with the above classifications is that there are usually many levels of abstraction at which to view complex systems, in addition to the choice of the number of phases. What is a design problem may be viewed as an implementation problem or an operation problem at another level, and *vice-versa*. This can make some comparisons across projects difficult. Therefore, this classification is subject to an abstraction level that needs to be specified. For the sake of auditing vulnerabilities, reasonable and useful but fuzzy definitions were provided in [3]. The time of introduction of a vulnerability in the SDLC has been used as a taxonomic characteristic [5, 6, 7] but was later shown to fail scientific requirements [8,9]. Some flaws can be introduced at multiple points in the SDLC, so although linking a vulnerability to a specific SDLC phase can be an interesting consideration, it is not always applicable.

This kind of classification may also be useful when reviewing software development processes in a software development firm. The details of the classification are then dependent on the specific software engineering model adopted. Therefore, there can be as many taxonomies of this kind as there are software engineering development models. This makes the comparison and discussion of vulnerabilities across development models difficult.

Classification by Genesis

Vulnerabilities, as flaws, can be classified into intentional and inadvertent flaws. Intentional flaws are further divided into malicious and non-malicious ones [2, 6]. However, the necessary information can be difficult to obtain. There is no procedure that can be followed to determine the above, and therefore the classification is not objective and not specific [9]. It has been argued that despite this limitation, tools attempting to catch specific flaw types are successful, so the classification is useful [2]. In effect, the classification helps understand which flaws can be caught by code scanning tools.

Classification by Location in Object Models

These classifications attempt to categorize vulnerabilities according to which model object or “entity” they belong to. Examples are classifying vulnerabilities using the ISO Open Systems Interconnect (OSI) reference model for networking [10]. This model defines 7 distinct layers, so a vulnerability could in theory be classified according to which layer it belongs to. In reality, some vulnerabilities depend on the specific interactions between two or more layers or objects, so this kind of taxonomy is not always applicable.

Similarly, attempts to differentiate whether a vulnerability was due to an operating system or an application were unsuccessful (CVE-1999-0144). This was because a configuration option of the operating system could prevent it from happening, whereas others argued that well-behaved programs shouldn’t need that kind of OS configuration. This difficulty can be encountered whenever the vulnerability relates to ill-defined responsibilities. The classification may then

change depending on the perception of the responsibilities and is therefore subjective. For example, currently the security of web browser plugins and scripting engines is considered their own problem. However, as web browsers mature, the better browsers should limit the security risks posed by vulnerable plugins and malicious scripts, in a manner similar to operating systems limiting what processes can do.

Classification by Affected Technology

Format string vulnerabilities in "C" are an example of a vulnerability type identified by the underlying technology. The implementation of polyvariadic functions in "C" is such that called functions have no way of knowing how many arguments were passed. Attacker-controlled format strings can exploit both that limitation and format string functionality, e.g., to write data at arbitrary locations in memory by using the "%n" format specifier. Depending on the format string supplied, the result of the attack can be: a) a crash by trying to access inexistent memory or memory allocated to another process (denial-of-service); b) the formatted strings contain unexpected data, possibly exposing confidential data, or adding incorrect or malicious information; or c) the process becomes under the control of the attacker (compromise).

Meta-character vulnerabilities happen in technologies where some characters have a special significance, e.g., syntactic. Meta-characters are often used to separate data and commands in dynamic query languages. "SQL injection", "LDAP injection" are example sub-categories of meta-character vulnerabilities. Character encoding issues are another example sub-category, inasmuch as special meta-characters indicate special encodings (e.g., URL "percent" encoding).

Resource exhaustion vulnerabilities happen when limited computer resources can be abused maliciously in a way that limits the functionality of the system, possibly resulting in a denial of service. Examples are SYN-flooding attacks resulting from the TCP protocol requiring memory for every connection request; memory leaks (specific to languages without garbage collection); and algorithmic complexity issues where an attacker is able to trigger worst-case behaviors in vulnerable algorithms.

Whereas the name of the affected technology is a useful and descriptive reference, there are many vulnerabilities that do not relate directly to a specific technology. Therefore this classification scheme is not universally useful.

Classification by Errors or Mistakes

Errors known to have led to vulnerabilities have been categorized by their cause, the nature of their impact, and the type of change or fix made to remove the error [11]. A similar type of classification is used often in "19 Deadly Sins", with categories such as "use of weak password-based systems", "failing to store and protect data securely", etc… [1]. Another example is the "double-free" memory management mistake.

This has educational value, especially when the mistake can be abstracted and applied to new situations. Unfortunately, there are situations in which any of several changes in different code locations, modules or even different programs altogether can fix a vulnerability, or at least block the known exploitation paths. Therefore, as a classification it can be ambiguous.

Classification by Enabled Attack Scenario

Sometimes the set of vulnerabilities that enable a specific kind of attack is highly descriptive and fairly precise. For example, "cross-site scripting" ("XSS") is an attack scenario, and "XSS vulnerabilities" are vulnerabilities enabling the injection of scripting code into content served to web browsers. They enable other attacks, but "XSS vulnerabilities" captures a unique common property and is a useful and succinct reference. It should be noted that many XSS vulnerabilities, but not all, are the result of meta-character handling vulnerabilities.

On the other hand, referring to "denial-of-service vulnerabilities" is not useful because denial of service is the consequence of an attack, and not an attack scenario, and can be achieved in many disparate ways, e.g., buffer overflows, format string vulnerabilities, etc…

A classification of vulnerabilities in network protocols proposed by Pothamsetty and Akyol [12] is interesting because it offers simultaneously a "test" or attack taxonomy and countermeasures. Even though the mistakes ("vulnerability classes") are presented first, it is evident that they are defined based on the attacks they enable. This kind of enumeration is likely to be proved incomplete as a new kind of attack may be uncovered in the future. However, it may still be useful in practice. The categories are:

1. Clear Text Communication
2. Non-Robust Protocol Message Parsing
3. Insecure Protocol State Handling
4. Inability to Handle Abnormal Packet Rates
5. Vulnerability Arising From Replay and Reuse
6. Protocol Field Authentication
7. Entropy Problems

The "test" or attack techniques are [2]:

1. Packet Sniffing (PS)
2. Protocol Field Fuzzing (PFF)
3. Protocol Field Spoofing (PFS)

[2] The attack techniques have been re-ordered to roughly match the order of the vulnerability types.

4. Packet Flooding (PF)
5. Replay (RP)
6. Reuse (RU)
7. Packet Size Variation (PSV)
8. Packet Number Variation (PNV)
9. Out-of-sequence Packets and Out-of-range Values
10. Special and Reserved Packets (SRP)
11. Information Retrieval[3]
12. Communication Initiation (CI)
13. Communication Termination (CT)
14. Encryption and Random Number Check (EC)

CLASP Classification

CLASP (Comprehensive, Lightweight Application Security Process) is a set of activities aiming to improve security [13]. It uses a classification focused on enumerating errors, but simultaneously includes conditions resulting from mistakes, as well as events. It has the following categories at the top level:

- Range and Type Errors. This includes the "write-what-where condition" as well as buffer overflows and "format string problems".
- Environmental Problems. This includes events such as the failure of a random number generator.
- Synchronization and Timing Errors. For some reason this includes statistical attacks. It also includes "Capture-replay", the vulnerability to which is usually a protocol flaw.
- Protocol Errors. This includes using a broken or risky cryptographic algorithm.
- General Logic Errors. This is a catch-all that includes things as diverse as using a "non-cryptographic" random number generator, or too few parameters being passed to a function (e.g., format string issues in "C").

Whereas the list of things that can go wrong is interesting, this classification has several flaws from a scientific perspective. A vulnerability may simultaneously be classified in several categories at once, for example if several mistakes are linked to the vulnerability simultaneously, or if a mistake results in a condition, or if an event triggers a condition. This classification is inconsistent when used at different abstraction levels. For example, when considered at a low level, a problem may be due to an integer overflow problem. At a higher abstraction

[3] This refers to checking the protocol for exposures, not vulnerabilities.

level, it may become the failure of a random number generator. Also, similar issues, e.g., cryptographic issues, are not grouped together. This may not matter in practice, as the goal of this classification is to discuss which kinds of vulnerabilities may be found during various activities.

Seven Kingdoms

This classification of software security errors has the following 8 top levels [14]:

1. Input Validation and Representation
2. API abuse
3. Security Features
4. Time and State
5. Error handling
6. Code Quality
7. Encapsulation
8. Environment. This category is mostly composed of configuration issues, that is, issues that do not belong in the first 7.

This classification simultaneously includes causes, consequences and bad practices. Therefore, a vulnerability can belong to several categories simultaneously, or be classified differently depending on the abstraction level used. The API abuse category, while an appealing concept, is especially ambiguous and conflicting with other categories. It harbors issues that could arguably be classified as input validation problems, such as a buffer overflow vulnerability caused by not performing input validation on potentially malicious names returned by a reverse DNS call [14]. Its main strength is that it makes sense when discussing the detection rules used by code scanning software. It can also help educating and conveying to programmers secure programming concepts, such as being aware of the rules involved ("contract") when calling a given API.

Classification by Exploitability

It has been argued that difference between flaws and vulnerabilities is that vulnerabilities are exploitable, whereas flaws are not necessarily exploitable [2]. Exploitability can be difficult to establish, therefore defining the scope of a taxonomy by including a criterium that is difficult to observe or deduce correctly is not a good idea. Yet the concept is useful, so I would like to introduce the classification of vulnerabilities depending on whether they are latent, potential or exploitable. A latent vulnerability consists of vulnerable code that is present in a software unit and would usually result in an exploitable vulnerability if the unit was re-used in another software artifact. However, it is not currently exploitable due to the circumstances of the unit's use in the software artifact; that is, it is a vulnerability for which there are no known exploit paths. A latent vulnerability can be exposed by adding features or during the maintenance in other units of code, or at any time by the discovery of an exploit path. Coders sometimes attempt to

block exploit paths instead of fixing the core vulnerability, and in this manner only downgrade the vulnerability to latent status. This is why the same vulnerability may be found several times in a product or still be present after a patch that supposedly fixed it.

A potential vulnerability is caused by a bad programming practice recognized to lead to the creation of vulnerabilities; however the specifics of its use do not constitute a vulnerability. A potential vulnerability can become exploitable only if changes are made to the unit containing it. It is not affected by changes made in other units of code. For example, a vulnerability could be contained in the private method of an object. It is not exploitable because all the object's public methods call it safely. As long as the object's code is not changed, this vulnerability will remain a potential vulnerability only.

Vendors often claim that vulnerabilities discovered by researchers are not exploitable in normal use. However, they are often proved wrong by proof of concept exploits and automated attack scripts. Exploits can be difficult and expensive to create, even if they are only proof-of-concept exploits. Claiming unexploitability can sometimes be a way for vendors to minimize bad press coverage, delay fixing vulnerabilities and at the same time discredit and discourage vulnerability reports.

Classification by Disclosure Process

Zero-day (a.k.a. "0-day") vulnerabilities are vulnerabilities that are unknown to the public, and presumably, to the vendors. The name comes from the notion of responsible disclosure, where vendors are notified first and given some time to fix the vulnerabilities before they are made public [15]. Zero-day vulnerabilities are made public without notifying the vendor ahead of time. Before disclosure, zero-day vulnerabilities may be sellable on black markets or some security companies (e.g., iDefense's "vulnerability challenges").

Following a 0-day disclosure, owners of the vulnerable systems are informed and may be able to take actions to mitigate the issue or to detect compromises. Simultaneously, attackers are informed of new ways to attack systems and vendors have to scramble to produce patches that will protect their clients.

Configuration issues, Exposures, System Vulnerabilities, "Proper" Vulnerabilities

For the purposes of this chapter, a system is a combination of interacting software and hardware, which may be located on one or several machines, and that performs a set of tasks as a whole. A system vulnerability may prevent a desired behavior, enable an undesirable one, or violate the integrity and correctness of a desired behavior, in reference to a specific security policy. This means that what constitutes a vulnerability in a system may not be considered a vulnerability when the same system is deployed in the context of a different security policy. Therefore, system vulnerabilities are not intrinsic properties of a specific software artifact; they are subject to an external security policy.

System vulnerabilities may exist even if a system is composed of software artifacts all with correct designs and all perfectly implementing their design. This

may be due to misconfigurations (e.g., the presence or execution of software contrarily to policy), designs that are inappropriate for the system, or to emergent properties due to the combination and interactions of software artifacts designed separately. This is the realm of compositional security. In particular, systems can be vulnerable to emergent abusive behavior attacks, in which legitimate acts can be combined to violate a policy [16].

The discussion of system vulnerabilities should reference the violated policies and be accompanied by a description of the relevant interactions with other software artifacts. Sometimes, the reasonable expectation that there is a relevant policy deployed somewhere is sufficient for the discussion of a system vulnerability, but the generic terms of that policy should be defined explicitly for the discussion. The MITRE Common Configuration Enumeration (CCE) effort aims to identify configuration issues by finding examples of a relevant "reasonable" policies, such as hardening configuration guides for operating systems [17].

"Proper" vulnerabilities are apparent when a specific software artifact is compared to its requirements and design (explicit or implied) at the time of its creation. It can be argued that the design and design goals of a software artifact are intrinsic properties of the artifact, because they remain the same regardless of where, how, when, or by who the software is used. The examination of software artifact vulnerabilities can therefore be performed with less information than needed with system vulnerabilities, while being more objective and reproducible, because it is grounded in technical facts proper to the artifact. However, this can become subjective if some aspects of the design are guessed by the researcher. The MITRE Common Vulnerabilities and Exposures (CVE) effort gives unique identifiers to software artifact vulnerabilities [18].

Understanding the difference between system vulnerabilities and software artifact vulnerabilities is useful when practitioners and academics attempt to communicate. A practitioner may argue that a system is not vulnerable due to a configuration that blocks attacks, while the academic will point out that a vulnerability remains (and could get exposed again). Likewise, a practitioner may consider a system to be vulnerable due to a service running and exposing excess information when it shouldn't, in violation of a policy. In this situation the academic may lose interest because the violated policy is "arbitrary", i.e., orthogonal to the design of all of the software artifacts.

Exposures are information leaks that may aid an attacker but do not directly violate the design goals of a software artifact. This distinction was at the origin of the name change of the CVE from "Common Vulnerabilities Enumeration" to "Common Vulnerabilities and Exposures", in order to be inclusive of unexpected exposures. As the CVE effort matured and the CCE effort was introduced, the CVE focused on exposures that were not part of the design of a software artifact. Exposures that can be easily prevented without disabling a useful feature are handled by the CCE.

Some vulnerabilities can be examined as both system and software artifact vulnerabilities; consider CVE-1999-0997. This vulnerability arose due to the interactions between wu-ftp with ftp conversion enabled, and compression programs such as tar. This can be discussed from the point of view of a system vulnerability, because it appears through the combination and interactions of various software artifacts. It can also be considered a software artifact vulnerability because wu-ftp was designed to work with compression programs through the conversion option. A difficulty in creating such programs is that a design can be suddenly invalidated by the introduction of a new feature in an interacting product, or by the introduction of a new product that supports unexpected features.

Popular Attack Classifications

Popular attack classifications often use a mix of ambiguous classifiers such as the origin, goal, mechanism and motivation of an attack, and suffer from a perspective ambiguity. The purpose of an attack may be clear to the attacker, but can appear as something else to the defender. The DARPA 1999 IDS evaluation program used these 5 types [19]:

1. Probe (or surveillance). This category applies to activity meant to gather information.
2. Denial of service. This is really the consequence of an attack.
3. R2L (remote to local), i.e., unauthorized access from a remote machine.
4. U2R (user to root), i.e., unauthorized transition to root for an unprivileged user
5. Data. This is meant to represent attacks whose goal is to obtain and extract ("exfiltrate") confidential files from a system.

Failed or misunderstood attacks could be put in the probe category by a defender. An attack that may have been meant to be a probe by an attacker may result in a denial of service, accidentally or not. Denial of service can also be the consequence of a failed R2L or U2R attack. Data attacks can also be R2L or U2R attacks, so an attack can be classified in two categories simultaneously [19]. These categories aren't complete, as they don't include U2U (user to user) attacks or take into account new attacks, for example attacks aimed at attacking other users of the system by planting malicious links or scripts (e.g., XSS attacks) or misinformation. Nevertheless, this classification was useful and appropriate for the IDS evaluation, given attacks with known goals.

Web Application Security Consortium Threat Classification

The WASC's threat classification has the following top classes:

1. Authentication. This identifies the mechanisms attacked (targets).
2. Authorization. This identifies the mechanisms attacked (targets).

3. Client-Side Attacks. This really is a technology type classifier, in this case used to identify a target.
4. Command Execution. This is a goal.
5. Information Disclosure. This is a consequence.
6. Logical Attacks. This contains as a sub-class denial-of-service attacks, which is a consequence. Other sub-classes describe mistakes, or attacks against access control mechanisms (which were covered in class 2).

In conclusion this classification uses inconsistent types of classification criteria at the same level, which leads to ambiguities, as an attack can be classified in several categories at once.

Classification by Transactional Attack Scenario

A transactional attack scenario describes how the attack operates on the basis of transactions between the participants. Examples are replay attacks, man-in-the-middle and the strictly-defined cross-site scripting (XSS) attacks that involve a third-party host. Eavesdropping is another transactional attack scenario that simply requires listening and passively gathering information. It can be carried out by listening to incidentally leaked signals (lights of LEDs, sounds of typing on keyboards, monitor radiation), or to network or wireless (including Bluetooth) transmissions (a.k.a. “sniffing”). These attack classifications are most useful when studying the security of communication protocols.

Impact

The impact of the attack on the confidentiality, integrity and availability of targets has been used by the Common Vulnerability Scoring System (CVSS) with levels of “none”, “partial” and “complete”. These coarse levels indicate the impact of the worst attack enabled by a vulnerability [20]. As the worst attack scenario may not be known yet (perhaps it is enabled only in some circumstances), this classification may be subject to change as more information is found, contrarily to the goal of the CVSS to use this as an invariant in the scoring system.

Attack Language

Attack instances can be described as a series of steps to achieve an unauthorized result [4]. This includes the tool used, the vulnerability targeted, events comprised of actions and targets, and finally an (at least attempted) unauthorized result. Tool categories are:

- Physical Attack
- Information Exchange. This includes social engineering attacks (see below).
- User Command
- Script or Program. This includes trojan horses (see below).

- Autonomous Agent. This includes viruses and worms (see below).
- Toolkit. This includes rootkits (see below).
- Distributed Tool
- Data Tap. This is the monitoring of energy leakage through an external device. This includes light (videos, pictures, blinking LEDs), variations in power consumption, sounds of keys being pressed, radio waves, etc…, which may reveal information.

Social Engineering Attacks

"Social Engineering" is a powerful descriptor of an attack that involves tricking human beings. The human being may not be the final target of the attack, but is being used by deceptive means. Social Engineering encompasses also deceptive computer-human interfaces (a.k.a. "phishing"), human interactions, phone calls to collect information from naïve employees, disguises and pretenses (a.k.a. "pretexting"). Some of these attacks are assisted by malicious code (see trojans).

Classification of Malicious Code

The classification of the malicious code used in an attack can be useful to understand the attack. However, for the sake of objectivity, code should be classified from its behavior alone, without needing to know that it was created with malicious intent (maliciousness may be inferred later, though). Indeed, benevolent viruses have been discussed before [21]. An attempt to create a benevolent worm, the Welchia worm, backfired [22], so intent can be largely irrelevant. I also object to the classification of viruses as a sub-class of Trojan Horse [5], since a virus may replicate without needing to deceive users by appearing as another code entity (e.g., boot sector viruses, or macro-viruses in formats supporting macro-scripting). It makes more sense to consider the need to deceive users separately from self-replicating aspects. According to the criteria proposed in [23], it is more useful and desirable to consider "primitive" classifiers that can each be answered by yes or no. I offer the following definitions of various aspects of malicious code:

1. *Attack Code.* Attack code aims at exploiting vulnerabilities, and is commonly found in the form of attack scripts or proof-of-concept exploits. Worms are another example of attack code. Malicious code isn't necessarily attack code, but its mere presence may imply that the system was compromised by a prior attack. Malicious code resident on a victim computer and performing an undesirable function, such as spyware, rootkits or backdoors, is to be differentiated from attack code that exploits vulnerabilities.
2. *Parasitic Code.* Parasitic code is code that is attached or included in another document or executable and violates its integrity. Intended or

original properties of the document or executable must be identifiable in order to determine the presence, nature and extent of the parasite. Parasitic code is not necessarily attack code.

3. *Back-doors (a.k.a. "Trapdoor" [6])*. A back-door is malicious code allowing the violation of user authentication mechanisms (so there is no doubt that it is malicious). For example, a remote user may issue commands as root through a previously installed back-door.
4. *Trojans*. Code that gets executed by deceiving a user is a trojan (the deception aspect implies maliciousness, even if it is a mild prank). Trojans can carry and be the initial entry mechanism for malicious code of another nature (e.g., a back-door or keylogger).
5. *Self-Propagating Code*. Self-propagation can occur in two modes:
    a. Viruses are parasitic and are spread by means of finding new host files (documents or executables) that will presumably also get read and run later. Macro viruses refer to viruses carried by documents which can carry "macros", essentially a scripting capability which blurs the boundary between data and code.
    b. Worms spread on their own, by duplicating their code to other systems and re-spawning their processes.
6. *Spyware*. Spyware is code that reports user activities and system information to "unauthorized" parties (who is "unauthorized" may depend on perspective). An example is an "unauthorized" keylogger. Spyware could also take "interesting" forms such as being a virus, and reporting when a certain type of document is opened.
7. *Logic/Time Triggered Code*. This is code that is not part of the normal function of a software artifact and remains dormant until very specific activation conditions are met. If it can then perform attacks, it is a "Bomb" [6]. If not, and it simply exposes humorous (at least to some) content, it is considered an "Easter egg". Triggered code that isn't a bomb or an Easter egg can be digital rights management ("DRM") code, if it enforces a licensing agreement.
8. *Rootkit*. A rootkit is a set of software artifacts that systematically deceives and misinforms a user whenever any part is executed (compare to trojan). Typically a rootkit replaces the utilities included with an operating system for the purposes of hiding a compromise and a back-door. A rootkit may include attack code as one of its components.
9. *Distributed Code*. Distributed code has coordinated copies of itself on many hosts. By acting in a coordinated fashion, the distributed code attempts goals that would likely be unreachable for a single copy. The coordination mechanism may be the reception of commands, or interactions between copies or with a controller. Blindly following specially crafted rules of conduct may also result in the overall desired behavior.

Worms have been known to include a time bomb for attacking a pre-determined target at a given time, resulting in a distributed attack.

Scientific Classifications

Scientifically correct taxonomies of vulnerabilities and attacks are difficult to create, due to the requirements that they separate elements of a group into subgroups (taxa) that are mutually exclusive. The crucial problem in designing a taxonomy is to identify and organize appropriate taxonomic characters, a.k.a. features, attributes or characteristics. The taxonomic characteristics must have these properties [9]:

1. *Objectivity.* Values must be based an observable property of the object.
2. *Determinism.* There must be a clear and explicit procedure to follow so that there is no possibility of misclassification.
3. *Repeatability.* Values must be selected reproducibly by different people.
4. *Specificity.* The selected value must be unique and unambiguous.

Finally the taxonomy must be exhaustive and useful for a broad audience. Many attempts have been made at classifying vulnerabilities and attacks, but most do not meet the above requirements. A typical error seen in the classifications previously discussed, is to use different types of taxonomic characteristics at the same level. This leads to ambiguities or nonsensical questions similar to asking “whether something breathes air or has eyes”.

Vulnerabilities are conceptual entities, not objects or lines of codes; this is an additional challenge to their understanding and classification, because desirable taxonomic characters should be observable without speculation. They exist at several different conceptual and abstraction levels; many vulnerabilities may exist simultaneously at several levels. In addition, the notions of cause or effect suffer from the problem that vulnerabilities may depend on several mishaps in a sequence of events. A "buffer overflow" may have been caused by an "integer overflow" issue. An off-by-one miscalculation resulting in a NUL-termination problem may enable the joining of two string buffers, which then provides sufficient space for a format string attack to be successful (CVE-2001-0609). There is also a possible decoupling of the coding mistake versus the exploit location. An overflow may first occur in a heap buffer but become (more easily) exploitable when the string is copied to a buffer of the same size on the stack (CVE-2006-0855); in this case the second copy is justifiably “correct”. Information about vulnerabilities is often incomplete and revealed progressively, which may change the "dominant" or primary aspect of the vulnerability or its "cause" as more details are learned.

Taxonomies that use criteria oriented towards defining what programmers, designers and architects have done incorrectly (implying what they should or

could have done) have had fatal flaws [24, 25, 26, 27]. The reasons why these taxonomies are incorrect were analyzed previously [8, 9]. Intuitively this can be understood because there are vulnerabilities whose exploitation could have been prevented or fixed by any of several different ways and mechanisms; therefore the classification of the vulnerability by the prevention measure or error is ambiguous. Typically these measures also involve different concepts and levels of abstraction. This situation is even expected when a vulnerability arises from an underspecified API [2].

Taxonomies also have a domain of validity and purpose, or scope and viewpoint, for example web services [28]. The specific data used to classify vulnerabilities depends upon the specific goals of the classification [23]. However, the scope and the taxonomic criteria should be technical in nature, to avoid speculation [23]. The classification of environmental assumptions proposed by Krsul [9] has limited usefulness, because the definition of its scope requires knowing the distinction between a designer misunderstanding the environment, and making a simplifying assumption about it. This may be difficult to decide, even in retrospect; it is not objective. So, it is not sufficient that taxonomic characteristics inside the taxonomy have the properties defined above; the scope of the taxonomy should also possess them.

Classification by Violated Program Invariant

Most vulnerabilities can be expressed in the form of an assumption, explicit or implicit, that didn't hold [9]. Some assumptions can be expressed as program or unit-wide invariants, others specifically as pre-conditions and post-conditions to algorithms, functions. All these properties need to hold true for the program to be correct. Some can be explicitly tested, e.g., by using "assert" statements. Some invariants that are not tested or stated can be discovered by dynamic analysis [29].

Buffer overflows are primarily the result of violating the invariant that data should only be read or written to within the space allocated for it. They are very common in "C" because the programming language makes the preservation of that invariant difficult and error prone. As a result, "buffer overflow" is a common vulnerability type that could belong to a taxonomy where vulnerabilities are classified by violated invariant or assumption. Such taxonomies have the potential to be very powerful and precise. Unfortunately, many assumptions are made unknowingly, and they tend to be unique, which results in a high cardinality of "types". For these reasons it is not useful to try to express all vulnerabilities as violated assumptions or violated program invariants. However, the explicit specification of invariants in programming languages such as Spark, where they get verified, can be quite useful in avoiding vulnerabilities [30].

Enumeration of attack and vulnerability types

As an alternative to producing a rigorous taxonomy from the top down, efforts have tried to tackle the enumeration of all known attack and vulnerability types

from a very low level. These enumerations in effect form both test cases and requirements for successful taxonomies. Due to their high cardinality nature, it is impossible to list the enumerated types in this chapter. What follows is merely an introduction to these efforts, pointing out their useful properties.

PLOVER

The Preliminary List of Vulnerability Examples for Researchers was a MITRE effort grouping CVE issues by low-level types of weaknesses [31]. It has been superseded by the CWE (see below). This was the first attempt to build an "a posteriori" classification, i.e., bottom-up ("a priori" classifications are built top-down, from a theoretical stand point).

Common Weakness Enumeration (CWE)

The CWE is a MITRE-driven effort to produce a "standard dictionary of software security weaknesses" [32]. The CWE attempted to be a community effort by enlisting industry, academia, and government. Amongst many benefits, it should allow the verification of coverage claims made by software security tool vendors and service providers [32]. At the top level, the CWE contains "location", "genesis" and "time of introduction" classifiers, which can be used independently, as is applicable and as knowledge permits.

The "genesis" classifier is subject to the limitations and objections noted before. The sub-categories for the time of introduction classifier are very detailed, considering events such as bundling and porting. Because the sub-categories are flat, a high level of knowledge is required to match the high level of detail in order to make a correct choice. Otherwise, classifications will be biased towards more generic-sounding categories such as "design" or "implementation".

The "location" classifier is sub-divided into Environment, Configuration and Code. Configuration issues are not further expanded upon, and should be the subject of further research (c.f., the MITRE CCE effort), especially when configuration options are so complex as to require their own language (is it code then?). The code sub-categories are subject to the criticisms appropriate to the classifications borrowed by the CWE. It is however very interesting to see how the PLOVER examples were mapped to those classifications.

Common Attack Pattern Enumeration and Classification (CAPEC)

CAPEC describes attack patterns and relates them to weaknesses in the CWE. It includes "not only weaknesses directly related to the attack, but also those whose presence can directly increase the likelihood of the attack succeeding or the impact if it does succeed" [33]. The CAPEC schema description is a work performed under contract for the Department of Homeland Security, and is therefore public.

Attack patterns aim to represent aggregate abstract information about similar attacks: “An attack pattern is a general framework for carrying out a particular type of attack, such as a method for exploiting a buffer overflow or an interposition attack that leverages certain kinds of architectural weaknesses” [34]. As such, they are a form of attack classification.

Ontologies and Future Research

The definition of a common language for computer security is an important step towards being able to describe and communicate vulnerability and attack knowledge [4]. Seacord and Householder observe that the classical taxonomy approach hasn’t worked well in security and suggest “a structured approach to classifying security vulnerabilities”. They do this by describing vulnerabilities and exploits as things with properties [35], which resembles part of an ontological approach to the problem.

Vulnerabilities are concepts, and many taxonomies attempt to link them to more concepts. Ontologies are models of reality in the form of a highly structured system of concepts, including their properties. So, in reality these “taxonomies” are attempts at creating partial ontologies. They present a reduced ("flattened") or simplified view of the accumulated knowledge, and may be appropriate for limited, specific purposes, as defined by their scope and goals. To completely and accurately represent, transmit knowledge and discuss vulnerabilities and attacks, proper ontologies are required.

Common ontologies address a clearly defined but restricted domain. The wider the domain, the more difficult the task may be. Upper ontologies are applicable across a wide range of domain ontologies, but are difficult to create. One reason why an upper ontology may succeed where taxonomies failed is that ontologies can be more flexible and complex. Ontologies may allow an object to belong to several classes at once, because partitions are not necessarily disjoint. Also, through the use of relations (e.g., "part-of") they may allow multiple "memberships" or inheritances to represent complex cases. I believe that an upper ontology created this way could have simpler "views" (e.g., subsets of the relations and concepts) appropriate to particular domains or applications. In practice, a number of regular ontologies with smaller domains could be created before the task of an upper ontology is addressed. They would also be easier to use. The development of such ontologies is left for future research.

References

[1]. Howard, M, LeBlanc, D and Viega, J (2005) 19 Deadly Sins of Software Security. Emeryville, CA: McGraw-Hill/Osborne

[2]. Weber S, Karger PA and Paradkar A (2005) A software flaw taxonomy: Aiming tools at security. Software Engineering for Secure Systems (SESS’05).

[3]. Dowd, M, McDonald, J and Schuh, J (2006) The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Boston, MA: Addison Wesley Professional

[4]. Howard JD and Meunier P (2002) Using a “common language” for computer incident security information. Computer Security Handbook, Chapter 3. Seymour Bosworth and M.E. Kabay, Editors. John Wiley and Sons publishers.

[5]. Piessens F, A (2002) Taxonomy of causes of software vulnerabilities in Internet software, Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering (Vouk, M., ed.), pp. 47-52,

[6]. Landwehr CE, Bull AR, McDermott JP and Choi WS (1994) A taxonomy of computer program security flaws. ACM Computer Surveys 26(3):211–254.

[7]. Bishop, M. (1995) A taxonomy of unix system and network vulnerabilities. Technical Report CSE-9510, Department of Computer Science, University of California at Davis.

[8]. Bishop, M, Bailey, D (1996) A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis.

[9]. Krsul IV (1998) Computer Vulnerability Analysis. PhD thesis, Purdue University.

[10]. ISO 7498:1984 Open Systems Interconnection - Basic Reference Model

[11]. Du, W, Mathur, AP (1998) Categorization of Software Errors that led to Security Breaches, In Proceeding of the 21st National Information Systems Security Conference (NISSC'98), Crystal City, VA.

[12]. Pothamsetty V, Akyol, BA (2004): A vulnerability taxonomy for network protocols: Corresponding engineering best practice countermeasures. Communications, Internet, and Information Technology 2004. St. Thomas, US Virgin Islands

[13]. Secure Software (acquired by Fortify). Comprehensive, Lightweight Application Security Process. Available at: http://www.owasp.org/index.php/ Category:OWASP_CLASP_Project

[14]. Tsipenyuk, K, Chess, B, and McGraw, G (2005) "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," IEEE Security & Privacy, vol. 3, no. 6, pp. 81-84.

[15]. Christey, S and Wysopal, C (2002) Responsible Vulnerability Disclosure Process. INTERNET-DRAFT "draft-christey-wysopal-vuln-disclosure-00.txt". The Internet Society

[16]. Wing, JM “A call to action” (2003) IEEE Security and Privacy, Nov-Dec., pp. 62-67

[17]. MITRE Common Configuration Enumeration (CCE) http://cve.mitre.org/cce/

[18]. MITRE Common Vulnerability Enumeration (CVE) http:// cve.mitre.org/

[19]. Lippmann, R, Haines, JW, Fried DJ, Korba, J and Das, K. (2000) The 1999 DARPA Off-Line Intrusion Detection Evaluation. Computer Networks 34(4) p. 579-595.

[20]. Chambers, JT and Thompson, JW (2004) Common Vulnerability Scoring System, tech. report, US Nat'l Infrastructure Advisory Council. Available at: www.dhs.gov/xlibrary/assets/niac/NIAC_CVSS_FinalRpt_12-2-04.pdf

[21]. Cohen, FB (1991) A Case for Benevolent Viruses. Fred Cohen &Associates, http://www.all.net/books/integ/goodvcase.html

[22]. Sophos plc., (2003) 'W32/Nachi-A', available at: http://www.sophos.com/virusinfo/analyses/w32nachia.html

[23]. Bishop, M (1999) Vulnerabilities Analysis. Web proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection (RAID'99), http://www.raid-symposium.org/raid99

[24]. Bisbey II, R., and Hollingworth, D (1978) Protection Analysis: Final Report; USC/ISI, Marina Del Rey, CA 90291

[25]. Neumann, PG (1978) "Computer System Security Evaluation,"1978 National Computer Conference Proceedings (AFIPS Conference Proceedings 47), pp. 1087–1095 (June 1978).

[26]. Abbott, RP, Chin, JS, Donnelley, JE, Konigsford, WL, Tokubo, S, Webb, DA (1976) Security Analysis and Enhancements of Computer Operating Systems National Bureau of Standards. Accession Number : ADA436876

[27]. Aslam, T (1995) "A Taxonomy of Security Faults in the UNIX Operating System," Master of Science thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907 .

[28]. Vanden Berghe, C, Riordan, J, Piessens, F (2005) A Vulnerability Taxonomy Methodology applied to Web Services, Proceedings of the 10th Nordic Workshop on Secure IT Systems (NordSec). Lipmaa, H. and Gollmann, D., eds., pp. 49-62, 2005

[29]. Ernsty, MD, Czeislery, A, Griswoldz, WG, and Notkiny, D (2000) Quickly Detecting Relevant Program Invariants. ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering, (Limerick, Ireland), June 7-9, pp. 449-458.

[30]. Barnes, J (2003) High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley.

[31]. Christey, S. PLOVER: Preliminary List Of Vulnerability Examples for Researchers. Available at: http://cve.mitre.org/docs/plover/plover.html

[32]. Martin, RA, Christey, S, Jarzombek, J. The Case for Common Flaw Enumeration. NIST 2005 Workshop on "Software Security Assurance Tools, Techniques, and Methods", Long Beach, CA., USA

[33]. Barnum, S. Common Attack Pattern Enumeration and Classification (CAPEC) Schema Description (U.S Department of Defense under Contract HSHQPA-05-A-00035). Dulles, VA: Cigital, Inc.

[34]. Barnum, S, Sethi, A. (2006) Introduction to Attack Patterns. Available at: https://buildsecurityin.us-cert.gov/ daisy/bsi/articles/knowledge/attack/585.html ?branch=1&language=1

[35]. Seacord, RC, Householder, AD (2005) A Structured Approach to Classifying Security Vulnerabilities. Technical note CMU/SEI-2005-TN-003