Unforgivable Vulnerabilities

Steve Christey
The MITRE Corporation
coley@mitre.org
August 2, 2007

Abstract

For some products, it's just too easy to find vulnerabilities. First, find the most heavily used functionality, including the first points of entry into the product. Then, perform the most obvious attacks against the most common types of vulnerabilities. Using this crude method, even unskilled attackers can break into an insecure application within minutes. The developer likely faces a long road ahead before the product can become tolerably secure; the customer is sitting on a ticking time bomb. These "Unforgivable Vulnerabilities" act like canaries in a coal mine. They are beacons of a systematic disregard for secure development practices. They simply should not appear in software that has been designed, developed, and tested with security in mind.

This paper highlights the most common of the unforgivable vulnerabilities, followed by a proposal that the research community establish a set of Vulnerability Assessment Assurance Levels (VAAL) that can be used as indicators for the relative security of software products. Unforgivable vulnerabilities would be exemplars of VAAL-0.

Background

With the number of publicly reported vulnerabilities steadily increasing year by year [Lemos] [Martin2007a], it seems like there's no end in sight. The same high-profile software continues to be battered. There is now enough history to try to compare products based on the number of disclosed vulnerabilities [Jones] [Cox], but these crude metrics have a number of important limitations, such as the varying skills of the researchers who analyze these products; the differences in disclosure practices by vendors; inadvertent or intentional biases by researchers and the tools they use; the continuing rapid advances in analysis, such as the discovery of new vulnerability classes and the growth of fuzzing; and the lack of consistent details in public advisories [Christey2006a].

By its very nature, high-profile software is going to be audited extensively and relentlessly; more people are going to pay attention to it, including the most skilled analysts, who might need to push the boundaries to find new problems. Consumers might be tempted to consider lower-profile alternatives with a less colorful disclosure history, but are the alternatives actually any safer? How do we know if we're actually

making any progress? How can the average consumer sift through vendor claims or make sense of the technical details?

Metrics for software assurance are still in their infancy, as evidenced by the strong emphasis on statistical analyses based on published vulnerabilities. However, we are likely to see significant gains in the next year or two. Some security-relevant metrics are being developed that evaluate a product on a macro level, typically using automated tools. Howard, Manadhata, and Wing have formalized the "attack surface" [Howard2003] [Manadhata2005] as a means of evaluating software security quantitatively, most recently by comparing the attack surface of two FTP servers [Manadhata2006]. While it seems likely that the attack surface is correlated with the number of vulnerabilities for an application, this has not been sufficiently proven, and one could construct pathological cases that would demonstrate that this correlation is not universally true. Traditional software metrics such as code size and complexity might also be useful in assessing security, but more work is needed here, too. Threat modeling methodologies such as Trike [Saitta] could provide a basis for other metrics but are still in the developmental stage. Commercial research firms use their own metrics when presenting results to their customers, but these are not publicly shared, often because of restrictions imposed by non-disclosure agreements, and are likely to be tool-specific. In the next year or two, additional software assurance metrics are expected to be introduced, such as the Security Quality Score announced by Veracode in June 2007 [Wysopal], or upcoming Web Security Evaluation Criteria by OWASP [Curphey].

Unfortunately, these metrics assume that a comprehensive analysis can be performed, typically requiring full access to the code, using high-quality tools with reduced (or at least quantifiable) false positive/negative rates, and significant labor. Some of these metrics are only appropriate for assessing implementation, not design. Universal application of these metrics to all products is impossible, and most consumers will not be able to conduct these analyses themselves. In many cases, an evaluation is likely to be unpublished.

Clues are still available, however, in the extensive information about the published vulnerabilities for many products. While comparing the raw number of vulnerabilities has its limitations, the same raw data can be analyzed along other dimensions, giving a more comprehensive view of the security posture of a specific product. The Common Vulnerability Scoring System [CVSS] uses multiple fields for evaluating the overall risk of an individual vulnerability. Some of these fields, such as Access Vector, Access Complexity, and Authentication, can be useful in understanding how much effort an attacker needs to exploit a vulnerability, but these fields are not very fine-grained. They do not capture the relative difficulty and complexity of the associated vulnerability, which can be an indicator of the maturity of a product relative to its vulnerability history.

The Typical Vulnerability History of a Product

Based on our CVE work through the years, covering over 25,000 vulnerabilities, we regularly see a product (or product class) follow these phases, as new vulnerabilities are discovered and resolved:

1) Obvious vulnerability types in critical functionality, such as buffer overflows in the username or password of a login feature
2) Incomplete fixes for the first vulnerabilities discovered in the product, often involving a rushed patch for a single input or parameter, while other closely related vectors remain unprotected
3) Susceptibility to variants of the common vulnerability types, often bypassing a simple protection scheme that only spots the most common manipulations [Christey2006b]
4) Common types that are restricted to particular environments, rarely-used functionality, or users with special privileges.
5) Elimination of the most common vulnerability types, typically involving a systematic code analysis and/or refactoring such as input validation frameworks
6) Susceptibility to more novel or rare vulnerability types and attacks, which often are not easily detectable by common tools or informal manual analysis
7) Unique vulnerability types that usually require expert analysis and extensive effort to find, possibly requiring a new attack or vulnerability class

These phases are roughly linear, although their order can vary slightly. There are likely additional phases that are rarely (if ever) seen in practice beyond phase 7.

We have seen this pattern repeated many times. Different components of the same product, or different product classes, can be in completely different phases. For example, high-profile web server software is in Phase 6 or 7, whereas many VoIP and ActiveX technologies are in Phase 1 or 2. Many file-format vulnerabilities involve classic buffer overflows, but since this product class was largely ignored until the past few years, file-format processors are still in the early stages of their vulnerability history. On the other hand, the security implications of integer overflows are still unknown to many developers, and they aren't necessarily as easy to find for a moderately skilled researcher. So, it is not a surprise that even high-profile software is susceptible to them [Martin2007a].

Of course, a poorly developed product with early-phase vulnerabilities is also likely to have later-phase vulnerabilities at the same time. However, the disclosure patterns for a product typically follow these phases in near-sequential order, which reflects the haphazard and incomplete nature of the techniques used by many members of the applied vulnerability research community.

Criteria for an "Unforgivable Vulnerability"

Given the previous observation of the phases of a product's vulnerability history, one could use the following criteria for determining if a vulnerability is unforgivable:

1) **Precedence.** Many other developers have made the same mistake in the past, including high-profile vendors, as reflected in press reports, vulnerability databases, and mailing lists. A developer who is actively thinking about security is likely to monitor these information sources periodically.
2) **Documentation.** Many white papers, books, articles, or other educational materials highlight the problem. With the numerous Internet resources and about a dozen books related to secure development, a developer who is interested in security would be able to find basic information easily, quickly, and often for free.
3) **Obviousness.** The issue is immediately obvious if considering possible attacks.
4) **Attack Simplicity.** The attack uses the most common manipulations for the associated vulnerability type, such as a long string of "A" characters for a buffer overflow.
5) **Found in Five.** The issue could be found within five minutes of limited, typically manual testing or code review. For example, many PHP remote file inclusion (RFI) vulnerabilities can be found using simple regular expressions that search for the use of variables in *include()* or *require()* statements. This criterion has the effect of emphasizing issues that are found in the most commonly used features of a particular product.

If a vulnerability discovery seems to match all five criteria, then it is highly suggestive of the developer's lack of security awareness (items 1 and 2) combined with a lack of security testing (items 3, 4, and 5). A vulnerability might be more "forgivable" if it does not satisfy some of these criteria. For example, some well-known vulnerability types such as buffer overflows, directory traversal, and XSS can have highly complex manipulations that bypass otherwise-solid protection schemes that are designed to prevent those issues. Also note how the amount of awareness of a vulnerability changes with time. While format string vulnerabilities are well-known in 2007, they were quite rare in 2001, so even if they were found in five in 2001, they would not have qualified as "unforgivable" at the time. Today, of course, is a different story.

For this paper, we will define an Unforgivable Vulnerability as one that matches Criteria 1 and 2, and at least two additional criteria.

The Lucky 13

Given the above criteria, following are some candidates for unforgivable vulnerabilities that satisfy all (or most) of the criteria for an unforgivable vulnerability. They are roughly ordered based on their frequency within CVE [Martin2007a].

1) Buffer overflow using long strings of “A” characters in:
    a. username/password during authentication
    b. file or directory name
    c. arguments to most common features of the product or product class
2) XSS using well-formed SCRIPT tags, especially in the:
    a. username/password of an authentication routine
    b. body, subject, title, or to/from of a message
3) SQL injection using ' in the:
    a. username/password of an authentication routine
    b. “id” or other identifier field
    c. numeric field
4) Remote file inclusion from direct input such as:
    a. include($_GET['dir'] . "/config.inc");
5) Directory traversal using "../.." or "/a/b/c" in “GET” or “SEND” commands of frequently-used file sharing functionality, e.g. a GET in a web/FTP server, or a send-file command in a chat client
6) World-writable critical files:
    a. Executables
    b. Libraries
    c. Configuration files
7) Direct requests of administrator scripts
8) Grow-your-own crypto
9) Authentication bypass using "authenticated=1" cookie/form field
10) Turtle race condition - symlink
11) Privilege escalation launching "help" (Windows)
12) Hard-coded or undocumented account/password
13) Unchecked length/width/height/size values passed to *malloc()*/*calloc()*

Discussion

The buffer overflow examples are straightforward. While buffer overflows continue to plague software, the associated manipulations are usually more complex than a long string of “A” characters, at least for products with a long history.

Some might argue that XSS is not immediately obvious, since it involves attacks in which the victim is another user instead of the server. Many others might argue that all XSS is unforgivable. However, it’s been well-known for 5 years and is a staple of any basic primer on web application vulnerabilities. XSS probably has a larger attack surface than any other vulnerability type, and such a wide variety of attack manipulations that an entire book is dedicated to the topic [Fogie]. XSS using the IMG tag is omitted from the unforgivable category because the developer is more likely to want to support IMG tags, and less likely to realize that javascript: URIs can be used. Certainly, the novelty and complexity are still quite low.

A “turtle race condition” occurs when the timing window is so large that a turtle could win the race. This occurs when the product does not perform any checks to establish

whether an action is safe. Many symlink following issues have this large window, by simply opening up a file without checking the existence or ownership of the file first. While many symlink issues happen with smaller timing windows (e.g. between an existence check and an fopen), such a check demonstrates that the developer is paying some attention to the environment.

Privilege escalation by launching "Help" on Windows-based systems is a frequently-documented attack that demonstrates that the product is not dropping or minimizing privileges, even for non-critical features that are assumed to be benign. However, the canonical attack of navigating to "cmd.exe" might not be obvious to most.

The "grow-your-own-crypto" vulnerability contradicts advice that has been published for years. Like hard-coded accounts, this is not necessarily found in five.

The direct requests and authentication bypass issues demonstrate a fundamental lack of understanding of the stateless, client-controlled nature of web technology, and the ease with which it can be subverted. These have been featured in the OWASP Top Ten and many articles and tutorials on web security.

For RFI, the given code example uses a GPC variable, which omits the role of register_globals entirely and demonstrates the programmer's assumptions that this GPC value could not be modified. Even when allow_url_fopen is disabled, such code is usually subject to obvious directory traversal attacks.

The use of unchecked values to *malloc()* and *calloc()* is often a precursor to buffer overflows, but it also illustrates a lack of attention to pathological cases that are not necessarily malicious in nature. However, its security relevance is still fairly new.

It could be argued that hard-coded accounts and passwords do not satisfy all the criteria. By their nature, they are not Found in Five. They continue to be used to this day, sometimes in high-profile software, despite the significant risk they pose.

There are probably other issues that satisfy all or most of the criteria for unforgivable vulnerabilities, so the Lucky 13 are not necessarily comprehensive. The community is encouraged to suggest additions.

Vulnerability Assessment Assurance Levels (VAAL)

David Litchfield originally proposed Vulnerability Assessment Assurance Levels (VAAL) as a useful way of communicating the extent to which a security analysis has been performed on a product [Litchfield2006]. It could also be useful as a guideline for conducting repeatable analyses of various depths. If fully developed and adopted, VAAL could be a boon to consumers, who generally do not have the skills or resources for assessing how secure a product is. It could be one element of a "Software Security

Facts" label, similar to food labels that provide nutrition and other information in a consumer-friendly format [Black].

Unfortunately, the research community has not explored VAAL further. While many researchers are independent or hobbyist, it is disappointing that professional research organizations have not pushed this topic much further.

Proposed Dimensions for VAAL

Following are some dimensions that could be considered for a VAAL score of a single vulnerability.

Assumptions of SDLC Maturity. The suggested dimensions make certain assumptions about the relative maturity of the developer with respect to the Secure Development Lifecycle (SDLC). CMMI and equivalent frameworks could inform this particular dimension [SEI]. SDLC maturity is not necessarily measurable in code. Levels might include:

1) No security awareness; or, awareness with little or no prioritization
2) Basic defenses against obvious attacks; grep analysis
3) Use of crude fuzzers and defenses for most vulns
4) Defense in depth; secure design that minimizes risk; independent review

Access Constraints. This considers what privileges or restrictions are needed for the vulnerability to be exploited. The developer might be expected to take extra precautions to protect features that are accessible to untrusted or unprivileged users. Remote and local vectors are intentionally excluded from consideration in this dimension, although many developers and consumers use them in prioritization.

1) Unauthenticated user or guest
2) Authenticated user
3) Authenticated user with special privileges
4) Administrator

Feature Frequency. This considers how frequently the affected feature/functionality is used within the product.

1) Universally used, which is always required during product interaction
2) Frequently used, which is not required during product interaction, but encountered frequently during typical usage
3) Rarely used
4) Non-standard

Potential Severity. This considers the amount of potential damage that could take place if a vulnerability is exploited. The developer might be expected to pay close attention to code or functionality that runs with extra privileges.

1) Root, Administrator, System
2) Other privileged user
3) Regular user
4) Restricted, guest, or anonymous user

Novelty. This considers how new or unusual the vulnerability and associated attack is. If we assume that software will always have vulnerabilities, then we might reasonably expect that it will not have vulnerability types that have been known for a long time.

1) Extensively documented, in "top ten" lists and book chapters
2) Well-documented in books (1 or 2 pages), white papers
3) Somewhat obscure or rare; below "top 30"
4) Brand-new class

Vector Depth. This considers the complexity of the code path that contains the vulnerability. It suggests how easily the issue could be identified using source code analysis. In vulnerability theory terms [Christey2007] [Martin2007b], this could be measured as a "distance" (in function calls or blocks) between the interaction and trigger points, i.e. where input enters the product and where the attacker actually triggers the vulnerability. The crossover and activation points might also be useful. For example, in XSS and second-order SQL injection, the activation point is in a different process than the trigger point, which would be less likely to be found during code analysis or conceived during basic threat modeling. Other useful aspects might include the actors and roles involved; for example, triggering a buffer overflow using long DNS server responses will typically involve a client, server, and consultant. Considering just the location of the Interaction, Crossover, Trigger, and Activation (ICTA) points, one might distinguish vector depth as:

1) ICTA all in a single function
2) ICTA all in a single file
3) ICTA all in a single process
4) Interaction and Activation in separate processes (i.e. second-order attacks)

Manipulation Complexity. This considers the inherent complexity of the manipulations being used in the attack. It could reflect the amount of external testing that the developer has conducted. More complex manipulations are typically required to bypass the developer's protection schemes or work within the constraints of unusual algorithms.

1) The canonical manipulation for the vulnerability type, such as <SCRIPT> for XSS, or "%s" for format string vulnerabilities.

2) Equivalence manipulations of the canonical manipulation, e.g. "%2e%2e" as an equivalent URL encoding for "..", or ".exE" as an alternate capitalization of ".EXE".
3) Compound manipulations
4) Manipulations that are technically malformed upon entry to the application, but are modified (e.g. by filtering) to become valid, typically a protection scheme failure.
5) Malformed or non-conformant manipulations that are treated as valid outside the product. These occur frequently in browser-specific XSS variants or in proxies and other intermediaries that might interpret a data stream differently than the product that ultimately processes that stream.

Ubiquity. This considers how much the affected feature is in use during operation of the product. Some issues might only show up in rarely-used configurations, or on 64-bit platforms but not 32-bit, or only when the product is deployed on a particular operating system.

1) Default configuration, all platforms
2) Non-default configuration, all platforms
3) Default configuration, some platforms
4) Non-default configuration, some platforms
5) Interactions with external components not controlled by the product

Level of Effort. This considers how much effort it takes the researcher to find the vulnerability. Unlike the dimensions that are inherent to the issue, which can be quantified in a repeatable fashion, the level of effort would vary widely depending on the individual researcher's skills and the capabilities of the tools being used. Tool performance is not quantitatively comparable yet, although the [SAMATE] and [CWE] projects are pushing in this direction. The amount of manual labor, however, is more easily quantifiable.

1) Less than 1 hour
2) Less than 1 day
3) Less than 1 month
4) More than 1 month

VAAL and Unforgivable Vulnerabilities

Using VAAL dimensions, the unforgivable vulnerabilities clearly demonstrate the lowest level of assurance. They have:

- low access constraints
- very high feature frequency
- very low novelty

- low manipulation complexity
- low level of effort

In some cases, there might be a high vector depth, but a developer with solid testing processes would be likely to find the issue due to the other characteristics such as low manipulation complexity and high accessibility. The potential severity will vary depending on the product's design. In most cases, these issue will have high ubiquity, but this is also product-dependent.

Future Work

This paper is intended to spark discussion in the research and consumer communities, and to encourage the development of metrics for software assurance. VAAL, or a variation of it, could also be useful in evaluating the skill levels and diligence of vulnerability researchers and code analysis tools. VAAL-based metrics could be devised for individual vulnerabilities, or collections of them.

Additional contributors are welcome and encouraged, since this work is a small effort within the CWE project.

References

[Black] Paul Black
"Strawman Software Facts Label"
(note: original idea credited to Aspect Security)
http://hissa.nist.gov/~black/softwareFacts.html

[Christey2006a] Steve Christey
"Open Letter on the Interpretation of 'Vulnerability Statistics'"
http://archives.neohapsis.com/archives/fulldisclosure/2006-01/0135.html

[Christey2006b] Steve Christey
"Blacklist defenses as a breeding ground for vulnerability variants"
http://seclists.org/fulldisclosure/2006/Feb/0040.html

[Christey2007] Steve Christey
"Vulnerability Theory"
http://cwe.mitre.org/documents

[Cox] Mark Cox
"metrics" posts on blog
http://www.awe.com/mark/blog/tags/metrics

[Curphey] Mark Curphey
"Assurance Levels for Web Security"

http://securitybuddha.com/2007/06/11/assurance-levels-for-web-security/

[CVSS] Common Vulnerability Scoring System
http://www.first.org/cvss/

[CWE] Common Weakness Enumeration
http://cwe.mitre.org

[Fogie] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, Petko D. Petkov
"XSS Exploits: Cross Site Scripting Attacks and Defense"
Syngress, 2007

[Howard2003] M. Howard, J. Pincus, and J. M. Wing
"Measuring Relative Attack Surfaces"
Proceedings of Workshop on Advanced Developments in Software and Systems Security, Taipei, December 2003
http://www.cs.cmu.edu/%7Ewing/publications/Howard-Wing03.pdf

[Jones] Jeff Jones
"Security by Numbers" blog
http://blogs.csoonline.com/blog/jeff_jones

[Lemos] Rob Lemos
"Security flaws on the rise, questions remain"
http://www.securityfocus.com/news/11367

[Litchfield2006] David Litchfield
"How secure is software X?"
Bugtraq, May 2006
http://archive.cert.uni-stuttgart.de/bugtraq/2006/05/msg00227.html

[Martin2007a] Steve Christey, Robert Martin
"Vulnerability Type Distributions in CVE"
http://cwe.mitre.org/documents/vuln-trends/index.html

[Martin2007b] Robert A. Martin, Steve Christey & Sean Barnum
"Being Explicit about Software Weaknesses"
http://www.blackhat.com/html/bh-media-archives/bh-archives-2007.html

[Manadhata2005] P. Manadhata and J. M. Wing
"An Attack Surface Metric"
http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-155.pdf

[Manadhata2006] P.K. Manadhata, J.M. Wing, M.A. Flynn, and M.A. McQueen
"Measuring the Attack Surfaces of Two FTP Daemons"

ACM CCS Workshop on Quality of Protection (QoP), Alexandria, VA, October 30, 2006.
http://www.cs.cmu.edu/~pratyus/qop.pdf

[Saitta] Eleanor Saitta, Brenda Larcom, Michael Eddington, and Stephanie Smith. "Trike"
http://dymaxion.org/trike/

[SAMATE] National Institute of Standards and Technology
"Software Assurance Metrics and Tool Evaluation project"
http://samate.nist.gov

[SEI] CMU Software Engineering Institute
"Capability Maturity Model(r) Integration (CMMI)"
http://www.sei.cmu.edu/cmmi/

[Wysopal] Chris Wysopal
"Software Security Weakness Scoring"
Metricon 2.0
http://securitymetrics.org/content/Wiki.jsp?page=Metricon2.0