Common Weakness Enumeration
A Community-Developed Dictionary of Software Weakness Types

Introduction to Vulnerability Theory

Document version: 1.0.1 **Date:** October 29, 2009

This is a draft document. It is intended to support maintenance of CWE, and to educate and solicit feedback from a specific technical audience. This document does not reflect any official position of the MITRE Corporation or its sponsors. Copyright © 2009, The MITRE Corporation. All rights reserved. Permission is granted to redistribute this document if this paragraph is not removed. This document is subject to change without notice.

Author: Steve Christey, Conor Harris (cwe@mitre.org)
URL: http://cwe.mitre.org/documents/vulnerability_theory/intro.html

Table of Contents

Disclaimer

Introduction

Despite the rapid growth of applied vulnerability research and secure software development, these communities have not made much progress in formalizing their techniques, and the "researcher's instinct" can be difficult to describe or teach to non-practitioners. The discipline continues to be more black magic than science. For example, terminology is woefully garbled and inadequate, and innovations can be missed because they are mis-diagnosed as a common issue.

MITRE has been developing Vulnerability Theory, which is a vocabulary and framework for discussing and analyzing vulnerabilities at an abstract level, but with more substance and precision than heavily-abused, vague concepts such as "input validation" and "denial of service." Our goal is to improve the research, modeling, and classification of software flaws, and to help bring the discipline out of the dark ages. Our hope is that this presentation will generate significant discussion with the most forward-thinking researchers, educate non-expert researchers, and make people think about vulnerabilities differently.

Status

This document's most recent major revision was in July 2009, reflecting ongoing work that dates back to 2005. This is the first update to the document since 2007, and several important concepts are identified and explained for the first time in this new revision.

The evolution of vulnerability theory is mostly occurring within the context of the Common Weakness Enumeration (CWE), a classification of almost 800 types and categories of weaknesses that lead to vulnerabilities such as buffer overflows, XSS, insufficient randomness, and bad permissions. The theory is being used to explain classification problems and train non-expert analysts about the vulnerability researcher mindset. Some terminology has made its way into CVE, and it has helped us to understand new vulnerability variants - especially by giving a vocabulary for efficiently explaining why the variants are new.

This draft is the most extensive text that is available on the topic.

Intended Audience

This document is for technically proficient vulnerability researchers and secure software development practitioners with a broad background in the area. The audience is assumed to be knowledgeable about a broad range of common software types and protocols, vulnerability types, their associated attacks, and the common countermeasures that fail to protect against vulnerabilities.

While the terminology is still evolving, much of it has stabilized and been actively used within CWE.

Basic Concepts

This section summarizes the key concepts that currently form vulnerability theory. Some concepts are further elaborated in subsequent sections.

Products, Behaviors, and Resources

A PRODUCT implements FEATURES by performing certain BEHAVIORS that operate on

RESOURCES.

- PRODUCT: a software package, protocol design, architecture, etc.
- FEATURE: a main capability offered by the product.
- RESOURCE: an entity that is used, modified, or provided by the product, such as memory, CPU, file, cookie, news article, or network connection.
- BEHAVIOR: an action that the product takes to provide a feature, or an action that a user performs.

For example, an FTP server may have two main features:

- upload files
- download files

The FTP server performs a behavior that establishes a connection to which it listens for incoming requests. When a client connects to the FTP server, the server verifies the user's identity (using an authentication behavior) then might accept a command to download ("RETRIEVE") a file resource. Subsequent behaviors include: establishing a separate data connection for sending the file; converting the file data from an internal representation to the representation type specified by the client; sending the data; and closing down the data connection.

Manipulations and Properties

A behavior performs MANIPULATIONS on resources by preserving or modifying PROPERTIES. For example, a base64-encoding manipulation might be applied to a binary file so that the file can be handled as ASCII; the resulting encoding has the property of being "equivalent" to the original file.

- PROPERTY: a security-relevant attribute of a behavior, data, code, or resource that must be transformed or preserved throughout operation of the product. For example, user input might initially contain arbitrary contents. After the system verifies that the input only contains letters and numbers, then the input might be treated as valid with respect to the property "alphanumeric." If the input is fed into an SQL query, then it might have a property of "trusted" with respect to the property "contains no SQL syntax characters."
- MANIPULATION: the modification of a resource by a behavior, typically to change the resource's properties. This is generally used in the context of software as it manipulates inputs and system resources to ensure that security properties are enforced. It can also be used by attackers to modify resources so that they do not have the expected properties.

Channels, Actors, Roles, and Directives (CARD)

Within the product, one or more ACTORS take on certain ROLES and perform DIRECTIVES that trigger behaviors. These directives are sent across CHANNELS. The possible combinations of actors, roles, and channels form a TOPOLOGY. For example, a common topology involves a User process being run by an Attacker that might connect through a channel (TCP port 80) to a web Service being run by a Victim, where the User gives the directive "Log me in" and provides username and password data.

- ACTOR: an entity (product, person, or process) that interacts with the software or with other entities that use the software. Types of Actors include: User, Service, Outsider, Consultant (e.g., DNS or RADIUS), Monitor/Observer (e.g., intrusion detection, log monitor, debugger), Intermediary (e.g., firewall, anti-virus, proxy). Some entities can encompass multiple types of

actors, e.g. an intrusion protection system that monitors for attacks and terminates connections if something suspicious occurs.

- ROLE: a pattern of behavior or interaction that is associated with a particular actor. Roles include Victim, Attacker, Bystander, Accomplice, and Conduit.
- DIRECTIVE: a request, command, signal, or other interaction from one upstream actor through a channel to a different downstream actor (typically a product), which typically causes the downstream actor to perform a desired behavior. Examples of directives are "Log me in as a user," "Retrieve file," "Change permissions," and "Exit system."
- CHANNEL: a resource that is used for sending directives and data between actors. There are many types of channels, including: Socket, Serial port, Signal (which is an implicit channel), Environment variable (implicit channel), Pipe, etc. "Alternate channels" are not the primary channels, but alternate ways of moving data or directives between actors. For example, the "Shatter" attack uses an alternate channel (the internal Windows messaging system) instead of the GUI.
- TOPOLOGY: the interrelationship between actors and roles in a single behavior chain. For example, in a typical web-based SQL injection scenario, a User (acting as an Attacker) sends a crafted request to a web application (acting as a Conduit) over one channel (TCP port 80), and the web server sends a SQL query over a different channel to a database server (acting as a Victim).

Security Policies, Control Spheres, and Protection Mechanisms

The product has a SECURITY POLICY (intended or implemented) that defines one or more CONTROL SPHERES that specify restrictions on which actors are allowed to access which resources or behaviors. A PROTECTION MECHANISM is a set of behaviors that is intended to enforce the boundaries of a control sphere, i.e., protect the product (or other actors) from attack.

- CONTROL SPHERE: a (possibly empty) set of resources and behaviors that are accessible to a single actor, or a group of actors that all share the same security restrictions. For example, a blog management program might have a control sphere that allows an "administrator" to post new blog entries, and a separate control sphere would allow "readers" to read blog entries but disallow them from posting new entries.
- SECURITY POLICY: a specification by the product or user that defines one or more control spheres for one or more actors. An example policy might be "only the administrator can directly invoke programs under /home/cwe/bin; only authenticated users may read files from /home/cwe/webroot." The developer of the product has an INTENDED POLICY that dictates appropriate behaviors, resources, and properties. The product itself has an IMPLEMENTED POLICY, which is the code's implementation of the intended policy. Ideally, the intended and implemented policies are the same; otherwise, a vulnerability may exist.
- PROTECTION MECHANISM: a behavior or set of behaviors that helps to enforce an intended security policy for the product (or an associated actor). This is also called a "control" or "countermeasure" in some communities. Examples of protection mechanisms include input validation, whitelists, blacklists, taint checking, stack overflow detection, etc. Protection mechanisms may be different than SECURITY FEATURES such as authentication, access control, cryptography, and privilege management. A mechanism might be implicit or explicit, depending on the layer. For example, canary-based stack overflow protection is added by a compiler, so it is implicit at the source code layer. If source code filters an output string against XSS, this is an explicit protection mechanism.

Attacks, Consequences, Weaknesses, and Vulnerabilities

While weaknesses are defined less precisely within CWE at this stage, a more precise definition is available that uses concepts from vulnerability theory.

- ATTACK: an attempt by an actor to violate the intended security policy, i.e., to access behaviors or resources that are outside of the intended control sphere for that actor. This is typically performed using manipulations of resources or behaviors that violate expected properties. An attack might require bypassing a protection mechanism. For example, a guest visitor to a web site might attempt to access administrative scripts by requesting them directly.
- ATTACKER: an actor who attempts to violate intended security policy, i.e., who attempts to gain access to behaviors or resources that are outside of the software's intended control sphere for that actor.
- CONSEQUENCE: a behavior that violates the intended security policy.
- WEAKNESS: a type of behavior that has the potential for allowing an attacker to violate the intended security policy, if the behavior is made accessible to the attacker. For example, an unbounded strcpy() call is a weakness, since it might be subject to a buffer overflow if an attacker can provide an input buffer that is larger than the output buffer.
- VULNERABILITY: a set of one or more related weaknesses within a specific software product or protocol that allows an actor to access resources or behaviors that are outside of that actor's control sphere, i.e., that do not provide appropriate protection mechanisms to enforce the control sphere.

Chains and Composites

Multiple weaknesses can be combined together to form CHAINS or COMPOSITES.

- CHAIN: a sequence of two or more separate weaknesses that can be closely linked together to form a vulnerability. For example, a chain may occur when a program encounters an integer overflow (CWE-190) when calculating the amount of memory to allocate, which causes a small buffer to be allocated and leads to a buffer overflow (CWE-120).
- COMPOSITE: a combination of two or more separate weaknesses that can create a vulnerability, but only if these weaknesses occur simultaneously. For example, a symlink following attack involves several component weaknesses including filename predictability (CWE-340), inadequate permissions (CWE-275), and a race condition (CWE-362).

Layering and Perspectives

Resources, behaviors, properties, manipulations, channels, actors, and directives can be described at different levels of abstraction, or LAYERS. In some cases, the focal point of a weakness or vulnerability depends on the PERSPECTIVE of the analyst.

The CWE content team believes that understanding of weaknesses and vulnerabilities might be significantly improved with proper modeling of layering and perspectives. However, this is a relatively new concept that requires further investigation.

There are three primary layers as currently defined within vulnerability theory:

- SYSTEM layer: resources include memory, disks, files, and CPU; behaviors include system calls and process execution.
- CODE layer: resources include variables, structures, sockets, handles, and strings; behaviors include assignments, function calls, and exception handling.
- APPLICATION layer: resources include cookies, messages, profiles, headers, and databases; behaviors include storing data, authentication, and sending messages.

A resource at one layer may be implemented using resources at lower layers. For example, an application-layer certificate might be read from a code-layer socket; the

issuer might be stored in a code-layer string that occupies system-layer memory. A pointer to the string is stored in a code-layer variable.

Behavior Layers

Layering can apply to behaviors, too. Consider the following C code, intended for a privileged program.

```
/* ignore group ID for this example */
old_id = getuid();
setuid(0);
AttachToPrivilegedDevice();
/* CWE-252 [code layer], CWE-273 [application layer] */
setuid(old_id);
filename = GetUntrustedFilename();
WriteToFile(filename, "Hello world");
```

The code intends to raise its privileges to root to access a special device, then drop the privileges back to the original user ID. It then writes to a user-supplied filename, and defers to the operating system's permission checks to ensure that the user can write to the specified filename.

In certain environments, it is possible for setuid to fail in certain situations, such as PAM failures or exceeded process limits. As a result, there is a weakness in the code.

At the code layer, the weakness is CWE-252 - an unchecked return value from a function. But at the application layer, the function is used to drop privileges, so CWE-273 also applies - improper check for dropped privileges.

Demonstration Code: Bug Barrel

To illustrate vulnerability theory in action, the following code will be used in subsequent sections. This code contains at least 10 weaknesses.

The intended functionality of the code is to support a web page that contains messages from various users, such as a guestbook or comment page. The code extracts a MessageType parameter from a request (line 2), constructs the associated filename (lines 3-5), opens the file, (line 6), and sends each line of the file to the requestor - as long as it does not appear to contain any scripting code (line 8).

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

A non-exhaustive list of weaknesses and vulnerabilities is:

- CWE-120: Classic Buffer Overflow (lines 2->4->10)
- CWE-23: Relative Path Traversal (2->4->6)
- CWE-79: Failure to Preserve Web Page Structure (XSS) (7->9)
- CWE-134: Uncontrolled Format String (7->9)
- CWE-476: NULL Pointer Dereference (6->7)
- CWE-20: Improper Input Validation (2 & 8)
- CWE-116: Improper Encoding or Escaping of Output (7->9)
- CWE-73: External Control of File Name or Path (2->4->6)
- CWE-404: Improper Resource Shutdown or Release (6->10)
- CWE-252: Unchecked Return Value (2->4->6)

Control Spheres

A CONTROL SPHERE is a set of resources and behaviors that are accessible to a single actor, or a group of actors that all share the same security restrictions. This set can be empty.

A product's security policy will typically define multiple control spheres, although this policy might not be explicitly stated. For example, a server might define several spheres:

- one sphere for "administrators" who can create new user accounts with subdirectories under /home/server/
- a second sphere that covers the set of users who can create or delete files within their own subdirectories.
- a third sphere might be "users who are authenticated to the operating system on which the product is installed." This control might be implicitly defined through OS-layer permissions.

Each control sphere has different sets of actors, resources, and allowable behaviors.

Weaknesses and vulnerabilities can arise when the boundaries of a control sphere are not properly enforced, or when a control sphere is defined in a way that allows more actors or resources than the developer or system operator intends. For example, an application might intend to allow guest users to access files that are only within a given directory, but a path traversal attack could allow access to files that are outside of that directory, which are thus outside of the intended sphere of control.

Some weaknesses related to control spheres include:

- CWE-610 Externally Controlled Reference to a Resource in Another Sphere

  The product uses an externally controlled name or reference that resolves to a resource that is outside of the intended control sphere.

  This is a high-level weakness that includes path traversal and symlink following, since the filename can be changed to reference a file that is outside of an intended directory.

- CWE-669 Incorrect Resource Transfer Between Spheres

The product does not properly transfer a resource/behavior to another sphere, or improperly imports a resource/behavior from another sphere, in a manner that provides unintended control over that resource.

This high-level weakness includes problems such as Unrestricted File Upload (CWE-434) and Download of Code Without Integrity Check (CWE-494). In these cases, the transfer of resources is intentional.

- CWE-668 Exposure of Resource to Wrong Sphere

  The product exposes a resource to the wrong control sphere, providing unintended actors with inappropriate access to the resource.

  This high-level weakness includes problems such as Information Leak (CWE-200) Insufficiently Protected Credentials (CWE-522), and Incorrect Permission Assignment for Critical Resource (CWE-732).

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

Based on the strcpy() in line 3, the programmer has an intended control sphere in which external requests are only intended to be allowed to read files that are stored under the "/home/cwe/" web document root. With path traversal (CWE-22), however, a "../abc" from line 2 would generate a filename of "/home/abc.dat" - which is outside of the intended control sphere.

Bug Barrel may have another intended control sphere in which only Bug Barrel users can add messages (through separate functionality that is not seen in the code above). If the associated messages file has world-writable permissions, then any local user could modify the messages file to change its contents, without going through the intended web interface. The weak permissions have defined a control sphere that is too broad because it includes unexpected actors (i.e., local users.)

The intended security policy of Bug Barrel may also include: "messages may not contain any scripting." The associated control sphere might be "a user can only add a message that can contain boldface, italics, or underlined text." These requirements are not explicitly stated anywhere; it is implicit in the protection mechanism at line 8, which tries to strip <script> tags. However, this action is incomplete, so the implemented security policy is "a line in a message may not contain '<script>' at the beginning."

There is also an intended control sphere with respect to buffer overflows. One line 4, the programmer assumes that the strcat() will not write outside the boundaries of fname. The original allocation of fname is not seen in this code. However, whether the allocation was performed on the stack or the heap, the associated size value (call it "SIZE") defined a control sphere for fname. If ftype can contain a string that is longer than fname, then a buffer overflow in line 4 can occur. The strcpy() would write to adjacent memory (and possibly adjacent variables), which is outside the intended control sphere

of the fname buffer. At a higher layer, there is another control sphere: only the explicit behaviors in the source code should be executed. If an attacker can leverage the buffer overflow for code execution, then the attacker's shellcode probably contains behaviors that were not in the intended control sphere as implied by the source code.

Protection Mechanisms

A protection mechanism effectively defines (or enforces) a control sphere. This may be a subset of the full control sphere for the software; for example, a protection mechanism might only be used to sanitize data, not to perform authentication. A protection mechanism is correct if its implemented control sphere is the same as the intended control sphere for that protection mechanism. A mechanism has a vulnerability if its implemented control sphere subsumes the intended control sphere. The mechanism is unnecessarily restrictive if it is a proper subset of the intended control sphere - effectively, it does not allow the user to do everything that the user is supposed to be able to do.

A SECURITY FEATURE is a protection mechanism for any actor that is not the product, typically the product's user. For example, cryptography may be used to protect the user's data, but it might not have any role in protecting the product's own control sphere.

Types of Protection Mechanism Failures (PMF)

When a weakness or vulnerability occurs, this is due to a failure to provide the appropriate protection mechanism. Each protection mechanism failure can be broadly categorized as:

- MISSING: the developer does not use a protection mechanism at all. This is seen in a high percentage of weaknesses as documented in CWE.
- INCORRECT: the developer uses a protection mechanism that attempts to provide some defense, but an error within the mechanism itself allows it to be bypassed. This can be due to a more general weakness within the mechanism itself.

In CWE, the "improper" term is used as a generalization of both "missing" and "incorrect." This is typically used in cases in which both "missing" and "incorrect" applies, or if the specifics are not known.

Note that Jeff Williams (OWASP, Aspect Security) breaks down protection mechanism failures by at least four types:

- Missing: the same as "missing" in vulnerability theory
- Ignored: a form of "missing" in which a protection mechanism is available, but the developer does not use it in all situations
- Broken: the same as "incorrect" in vulnerability theory
- Misused: a form of "incorrect" in which the developer attempts to use the protection mechanism but does not invoke it properly

These distinctions are important for understanding how developers can introduce errors while trying to address security concerns, and in improving development practices to avoid these problems in the first place. However, within CWE, these distinctions are rarely important when describing general weakness types, since the method of

introduction is not relevant to the behavior that has been implemented.

As of CWE 1.4, the term "insufficient protection mechanism" is used to describe situations in which the mechanism can vary in strength on a continuous or sliding scale, instead of a discrete scale. The continuous scale may vary depending on the context and risk tolerance. For example, the requirements for randomness may vary between a random selection for a greeting message versus the generation of a military-strength key. On the other hand, a weakness that allows a buffer overflow is always incorrect - there is not a sliding scale that varies across contexts.

Sanitization Techniques (VICES)

SANITIZATION is a general term to describe the process of ensuring that input or output has certain security properties before it is used. When used, the term could be referring to one or more of the following: filtering/cleansing, canonicalization/resolution, encoding/decoding, escaping/unescaping, quoting/unquoting, or validation.

The following types of sanitization may be used. There may be others that have not been covered yet.

- VERIFICATION: check if the input is already good
- INDIRECT SELECTION: use the input as a mapping to known-safe values
- CLEANSING: modify the input until it has the expected properties, typically by filtering (the explicit removal of dangerous or otherwise invalid elements).
- ENFORCEMENT BY CONVERSION: convert the input into a different, well-controlled representation. For example, in PHP, a common mechanism for avoiding SQL injection is to apply intval() to all numeric inputs, which guarantees that the generated value is a number.
- SANDBOXING: Rely on an external or implicit protection mechanism to provide enforcement. For example, a web application firewall might reject a suspicious request or modify the request before passing it to the application being protected.

The following terms are relevant to two or more of the types of sanitization listed above:

- ENFORCE: a general term, meaning to check or manipulate a resource so that it has a property that is required by the security policy. For example, the filtering of all non-alphanumeric characters from an input is one mechanism to enforce that "all characters are alphanumeric." An alternate method of enforcement would be to reject the input entirely if it contains anything that's non-alphanumeric.
- CANONICALIZATION: a behavior that converts or reduces an input/output to a single fixed form that cannot be converted or reduced any further. In cases in which the input/output is used as an identifier, canonicalization refers to the act of converting that identifier. For example, when the current working directory is "/users/cwe," the filename "../xyz" can be canonicalized to "/users/xyz."

Canonicalization may be used in verification, cleansing, and in some cases, as a means of enforcement by conversion.

Comparison is frequently used in both verification and cleansing. One common weakness is Insufficient Comparison (CWE-697), in which he software compares two entities in a security-relevant context, but the comparison is insufficient, which may lead to resultant weaknesses. This can occur in at least two different ways: (1) the comparison checks one factor incorrectly; or (2) the comparison should consider multiple factors, but it does not check some of those factors at all.

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

The value of ftype is not validated at all, so there is missing input validation (CWE-20). The lack of validation has at least three distinct consequences. Since there is no check on the length of ftype, the buffer overflow (CWE-119) in lines 2 to 4 is possible. Since there is no check of the contents of the ftype variable, a path traversal (CWE-22) attack is possible through lines 2, 4, and 6. In addition, there is no validation that the ftype variable constructed a pathname for a file that exists, leading to a NULL pointer dereference (CWE-479) in line 7.

For cross-site scripting (CWE-79), there is an attempt to perform input validation on line 8, but this is using an insufficient blacklist (CWE-184). This is an incorrect protection mechanism. For the same XSS problem, there is also a missing protection mechanism - no encoding of the output (CWE-116) from the printf call in line 9.

For the uncontrolled format string (CWE-134) on line 9, the exposure of this string to external input was probably entirely accidental due to the missing "%s" specifier. Because of this accident, there is no protection mechanism.

For both the buffer overflow and the uncontrolled format string, there may be implicit protection mechanisms available. For example, some implementations of printf() have reduced the risk of format string problems by removing support for the "%n" specifier, which provides fine-grained control to an attacker but is not used in most real-world code. For buffer overflows, canary techniques are available at the compiler layer (such as Microsoft's /GS compiler option), and hardware-layer features such as the non-executable (NX) flag can reduce the impact of overflow exploits.

Chains and Composites

Many weaknesses and vulnerabilities arise because of the interaction of multiple factors. As a result, inconsistencies in classification and terminology problems can occur, because people may be concentrating on different parts of the same problem.

A CHAIN is a sequence of two or more separate weaknesses that can be closely linked together within software. One weakness, X, can directly create the conditions that are necessary to cause another weakness, Y, to enter a vulnerable condition. When this happens, CWE refers to X as PRIMARY to Y, and Y is RESULTANT from X. For example, if an integer overflow (CWE-190) occurs when calculating the amount of memory to allocate, an undersized buffer will be created, which can lead to a buffer overflow (CWE-120). In this case, the integer overflow would be primary to the buffer overflow. Chains can involve more than two weaknesses, and in some cases, they might have a tree-like structure.

A primary weakness (or "root cause") is the first error in the code, after which things

start to go wrong. For example, when an off-by-one error prevents a null byte from being added to a string, which causes a buffer over-read, the off-by-one error is primary.

A resultant weakness includes behaviors that either (a) add to the problem, (b) fail to correct it, or (c) are only exposed because of the primary weakness. For example, a PHP program might have a primary weakness that uses extract($_GET) to overwrite global variables that were not intended to be mutable through a web request. As a result, the extract() might modify the variables after input validation has occurred, leading to resultant XSS, SQL injection, file inclusion, or other issues, depending on how the variable is being used.

While CWE primarily contains "implicit" chaining relationships, there are several chains that are so common that they were assigned their own CWE identifiers. These are called Named Chains. For example, CWE-691 covers the integer-overflow-to-buffer-overflow chain in the previous paragraph. CWE-690 covers chains in which an unchecked return value leads to a NULL pointer dereference.

Chain Example

```
1     #define MAX 200
2     int width, height, sz;
3     width = ReadUntrustedInt();
4     height = ReadUntrustedInt();
5     if ((width > MAX) || (height > MAX)) {
6        ExitWithError("bad width/height"); }
7     sz = width * height;
8     buf = malloc(sz);
9     mmov(buf, ImageData);
```

This code example attempts to allocate a buffer that is large enough to hold a graphical image, based on the width and height that has been specified by an untrusted actor. Line 5 contains two insufficient comparisons (CWE-697) because it does not check to see if width or height are negative. As a result of the improper check, an integer overflow (CWE-190) can occur in line 7 when width and height are both large-but-negative values. An insufficiently-sized buffer is allocated in line 8, which then leads to a buffer overflow (CWE-120) in line 9.

Composite

A COMPOSITE is a combination of two or more separate weaknesses that can create a vulnerability, but only if they all occur all the same time. One weakness, X, can be "broken down" into component weaknesses Y and Z. For example, Symlink Following (CWE-61) is only possible through a combination of several component weaknesses, including predictability (CWE-340), inadequate permissions (CWE-275), and race conditions (CWE-362). By eliminating any single component, a developer can prevent the composite from becoming exploitable. There can be cases in which one weakness might not be essential to a composite, but changes the nature of the composite when it becomes a vulnerability; for example, NUL byte interaction errors (CWE-626) can widen the scope of path traversal weaknesses (CWE-22), which often limit which files could be accessed due to idiosyncrasies in filename generation.

Note that while composites have been defined for as long as chains, they have not been studied as closely, and they are not as well-understood. One complication is that composites may have components at multiple layers.

Composite Example

```
1     $tmp = "/tmp/logfile.$$";
2     open($fh, ">$tmp") || die;
3     print $fh "starting task.\n";
4     DoTaskAndLogResults($fh);
5     print $fh "Done!\n";
6     close($fh);
```

This code example attempts to execute a task and log any results into a temporary log file. Line 1 generates a predictable filename that includes the process ID. The open() in line 2 opens the target filename for writing. Not only is the filename predictable, it is also in a directory (/tmp) that typically has world-writable permissions. Since the open() will succeed even if the file already exists, there is a race condition in which an attacker can create the target file before line 2 is executed. The target filename could be a symbolic link that points to a file that is owned by the victim who is executing the code. The timing window is probably large, because there is no check to see if $tmp exists before it is even opened.

As a result, at line 2, there are three simultaneous conditions:

(1) the creation of a file with a predictable name

(2) the creation of a file in a directory with insecure permissions, and

(3) a race condition.

Fixing any one of these component weaknesses would eliminate or significantly reduce the vulnerability. If a filename is unpredictable, then an attacker cannot reliably pre-create a symbolic link with the expected name. If the file is created in a directory with restrictive permissions, then the attacker could not create a symbolic link in the target directory. If the race condition is omitted - e.g. by using lower-level coding constructs that ensure the open() will only succeed if the file did not exist beforehand - then the open would fail if an attack is launched.

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

This code contains several chains. In line 6, an unchecked return value (CWE-252) can lead to a NULL pointer dereference (CWE-476) if the fopen() call in line 6 fails. Since ftype is under full control of the requester and can contain near-arbitrary content, exploitation of this problem is probably simple. In line 8, there is an incomplete blacklist (CWE-184) that allows cross-site scripting (CWE-79) attacks. Other chains are also present.

There is a subtle composite that involves insecure permissions (CWE-732), a missing protection mechanism, and symlink following. Assuming a UNIX-based permission

model, the web application user can effectively access any world-readable ".dat" file under the root directory, not just /home/cwe. There is no way to directly specify permissions that state "fopen cannot access files regardless of this directory." It may be possible to force this restriction by creating a chroot() jail, although it might not be realistic (or portable). In conjunction with the path traversal (CWE-22) through lines 2/4/6, a local user could launch a symbolic link attack (CWE-61) against the .dat file in order to access any file that is readable by Bug Barrel, but not by the local user. Thus the absence of a suitable protection mechanism for file containment effectively becomes a composite with external control of a filename (CWE-73) that become components of symbolic link following and path traversal weaknesses. Note that this perspective of path traversal is not well-developed.

Chains may also be possible, in the sense that a multi-stage attack could occur. For example, even if the program is changed so that the messages file is properly encoded on line 9, XSS might still be possible by using the path traversal weakness to reference another file that contains malicious script.

Properties

Properties can apply to resources, data, or behaviors, including code. These properties are often a factor in weaknesses and vulnerabilities.

- VALIDITY: The degree of conformance to data/behavior specifications. Examples include:
  - GET index.html (no version)
  - non-existent username
  - "<SCRIPT" without closing tag
  - inconsistent length/payload
  - incorrect sequence of steps
  - packet with length field inconsistent with actual length
  - "US3R" token when only A-Z characters are expected

- EQUIVALENCE: Whether two identifiers, inputs, resources, or behaviors have different representations, but are ultimately treated as being the same. Examples include:
  - "../.." == "%2e%2e/%2e%2e" (in URIs)
  - "filename.txt" == "FileName.txt." (in Windows)
  - $_GET['x'] == $_REQUEST['x'] (in PHP)
  - step equivalence: (A->B->C) == (A->C)
  - "/tmp/abc" and a symlink to /tmp/abc
  - "localhost" and "localhost.example.com" (fully qualified domain name)
  - 127.0.0.1 and "hostname.example.com" (IP address and its domain name)

- MUTABILITY: Whether the resource is expected to be modifiable, by whom, and when. Examples include:
  - format string: if a format string is mutable, then attackers can modify the format of output and possibly introduce buffer overflows. For most format string vulnerabilities, the intended format string should have been a constant, i.e., it was never intended to be mutable by any actor

- stack-based buffer overflow: violates non-mutability of the stack and adjacent variables (relative to the source code layer)
- PHP's register_globals often violates non-mutability of critical variables
- modification of cookies or hidden form fields often violates the programmer's expectation that cookies cannot be modified
- an unprivileged user's access to a mutex for a privileged process may introduce immutability when mutability is required

- ACCESSIBILITY: Whether the resource can be accessed, by whom, and when. Examples include:
  - mutex
  - permissions
  - "Shatter" attack

- TRUSTABILITY: Whether the resource can be trusted to have certain properties. For example, an input field to a PHP application can not be trusted to have any specific contents. After the input is converted to a number via a function such as intval(), it can be trusted to be a number; however, it cannot be trusted to have a specific numeric value until other checks are performed.
- UNIQUENESS: whether the resource or identifier is unique. For example, a session identifier is assumed to be unique.
- PREDICTABILITY: whether a certain property or state of the resource (or identifier) can be sufficiently predicted. For example, a random seed that is created from the system's current date is predictable. A cryptographically strong hash generates outputs that are not predictable.

There may be other types of properties that require further investigation, such as atomicity, temporal relationships, and completeness.

Note that an input can be valid but security-relevant. For example, "O'Reilly" is a valid last name, but the apostrophe "'" is a special character for SQL and would be invalid if not quoted properly.

Dimensions of Validity

The validity of resources, data, or behaviors can be assessed (or might be assumed) based on several possible dimensions.

- LEXICAL: whether the item is a well-formed token. Examples: "My", "name".
- SYNTACTIC: whether the item follows expected syntax and grammar rules. Examples: "My name is John." (valid) "John name my is." (invalid). HTTP response splitting attacks (CWE-113) occur because of the use of CRLF sequences to modify expected syntax (CWE-93), which in turn causes the request to be misinterpreted.
- DOMAIN: whether the item is valid within the domain of interest. For example, "red," "green," and "blue" are valid within the domain of colors; they are invalid within the domain of common first names. An unchecked array index (CWE-129) is invalid if it points to an index that is not within the boundaries of the associated array. The sentence "I understand an orange hammer," while syntactically valid, is probably not valid in most domains.
- RELATIONAL: whether two or more items are consistent with respect to their inter-relationships. For example, a message that contains a content length that is different than the actual length of the content does not have relational validity. The commands "USER [user]" and

"DOWNLOAD [filename]" may be valid from a syntactic and domain perspective, but they might not be relationally valid if the user is expected to provide a PASSWORD command before the DOWNLOAD.

Following are some examples of validity with respect to HTTP requests:

- lexical: G#T / HTTP/1.a (command and version number)
- syntactic: GET . HTTP?1.1 (separator between protocol and version)
- domain: GITCHY / ABCD/1.1 ("GITCHY" and "ABCD" not valid)
- relational: a POST request with a Content-Length value that is inconsistent with the actual length of the request body

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

For the uncontrolled format string (CWE-134) in line 9, the programmer probably left out a "%s" as a format string specifier to the printf() call. As a result of this omission, the "line" variable serves as a format string and violates the expected MUTABILITY property that "the first argument to printf is immutable."

For the filename to be accessed, there are at least two assumptions of validity when the fopen() is invoked in line 6:

- (a) fname points to a file that exists
- (b) fname points to a file under /home/cwe

An unchecked return value weakness (CWE-252) does not ensure that fopen() succeeds, which would imply the VALIDITY of the first property. A path traversal weakness (CWE-22) violates the second VALIDITY property in lines 2, 4, and 6. Path traversal is possible because an input of "../abc" for ftype would generate the filename "/home /cwe/../abc.dat" which is EQUIVALENT to "home/abc.dat". As a result of the path traversal, the targeted file is ACCESSIBLE even when the intended policy states that it should not be.

For cross-site scripting (CWE-79), the test in line 8 assumes that the printf() in line 9 is only using TRUSTED data. However, the test did not consider all possible inputs that provide EQUIVALENT scripting capabilities, such as "<sCrIpT>" which is equivalent to "<script>" because HTML tags are case-insensitive.

If the permissions for /home/cwe/ allow local users to create their own .dat files within that directory, then this may provide some MUTABILITY to the directory that the programmer did not expect. The creation of a symbolic link would then further violate

the ACCESSIBILITY of the file that is targeted by the symbolic link.

There is probably also an assumption of lexical validity in that the MessageType parameter only contains alphanumeric characters, but this is not clear from the listed code. Further, there might only be a handful of legitimate MessageType values that are part of the expected domain.

Simplified Error Handling Model

Many weaknesses, and the vulnerabilities they create, can be caused by errors and other exceptional conditions, as covered by Failure to Handle Exceptional Conditions (CWE-703) under the Research View (CWE-1000).

In a security context, the proper detection and handling of exceptional conditions serves as a protection mechanism by checking and enforcing properties of resources and behaviors.

Techniques for exceptional conditions vary widely, such as:

- the "errno" convention in C, in which a function returns a value that indicates an error and sets the errno variable to indicate which type of error occurred
- exception handling in Java, C++, Ruby, and others
- protocol-specific techniques such as the "404 not found" error message in HTTP

Despite the variety of algorithms, techniques, and implementations in use, many of these can be characterized as part of a simplified model of error handling:

- CHECK BEFORE: before the software performs a behavior to manipulate a resource, it checks to see if an exceptional condition already exists, or if the condition will definitely occur if the behavior is undertaken
- HANDLE BEFORE: knowing that the behavior will fail or otherwise be incorrect, the software modifies its execution so that the behavior is avoided, or the software changes the conditions so that the behavior will succeed
- CHECK AFTER: after a behavior is attempted, the software verifies that it performed as expected, or otherwise detects if an exceptional condition has occurred
- HANDLE AFTER: in some cases the behavior will still be undertaken, and the CHECK-AFTER will detect that an exceptional condition occurred. There may still be a chance for the software to maintain or establish control over its behaviors and resources.

Weaknesses in the Error Handling "Life Cycle"

Each phase of the error handling "life cycle" may generate different weaknesses that are captured in CWE. Initial investigation by the CWE content team suggests that many of these weaknesses are independent of whether the problem occurs in a "before" opportunity or an "after" opportunity.

Two of the main CWE entries are:

- CWE-754 Improper Check for Exceptional Conditions
- CWE-755 Improper Handling of Exceptional Conditions

The children of CWE-754 identify some of the lower-level weaknesses that may arise for an improper check, such as:

- CWE-252 Unchecked Return Value
- CWE-273 Improper Check for Dropped Privileges

Under CWE-755, lower-level weakness children for improper handling include:

- CWE-636 Not Failing Securely ('Failing Open')
- CWE-209 Error Message Information Leak
- CWE-390 Detection of Error Condition Without Action

Code Example: malloc()

Consider the following code:

```
void SayHello () {
  str = (char *) malloc (2048);
  if (str == NULL) {
     ReportError();
     exit(1);
  }
  strcpy(str, "Hello world!");
}
```

The behavior of malloc() is to return NULL when there is insufficient memory available to allocate a buffer. From the perspective of the SayHello() function that calls malloc(), there is no standard mechanism in C to perform a CHECK-BEFORE by seeing if memory is available before it attempts the allocation. Thus no CHECK-BEFORE/CHECK-AFTER is feasible, and SayHello() must simply call malloc() and see if it succeeded.

The CHECK-AFTER occurs in the conditional that compares str to NULL. The HANDLE-AFTER is associated with the block containing the calls to ReportError() and exit().

Notice that the programmer had several opportunities for improper error detection and handling. If there was no comparison with str (i.e., no CHECK-AFTER), then the strcpy() could result in a NULL pointer dereference. Even with a correct check to see if str is NULL, the program must alter its control flow. If there is no call to exit(), for example, then the strcpy() could still lead to a NULL pointer dereference.

Also note that at another layer - such as the definition of malloc() itself - there may be an implementation-specific CHECK-BEFORE behavior that ensures that there is enough memory. The HANDLE-BEFORE for malloc() in this case would be to return a NULL pointer to signal the erroneous condition.

Code Example: Authentication

Consider the following code:

```
$user = $_POST['user'];
$pass = $_POST['pass'];
$result = AuthenticateUser($user, $pass);
if ($result != SUCCESS) {
  printError("no can do");
```

```
        die;
}
ShowSecretFile();
```

Here, there is an authentication task. By its nature, authentication should not grant access to the secret file until it is certain of the user's identify. Thus, the authentication test is a CHECK-BEFORE, and the error message with the exit is the HANDLE-BEFORE.

Note that, at this layer of code, there is not an opportunity for a CHECK-AFTER or a HANDLE-AFTER to take place - if the program shows the secret file to an unauthenticated user, then it is too late to do anything about it.

Code Example: Exception Handling

Consider the following Java code:

```
try {
   doSomething();
}
catch (BadException e) {
   handleProblem();
}
```

With exception handling techniques such as try/catch, the declaration of a catch effectively defines a HANDLE-AFTER action for an exceptional condition. In these cases, at this layer, there often is no explicit CHECK-BEFORE, HANDLE-BEFORE, or CHECK-AFTER. The implementation of exception handling performs these tasks at a lower level - either in the method definition within doSomething, a lower-level method, or the Java virtual machine itself.

There may be a possibility of defining a CHECK-BEFORE/HANDLE-BEFORE behavior, such as NULL pointer dereference prevention. This is not always feasible.

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

For the handling of the messages file, there is a path traversal weakness (CWE-22) from lines 2, 4, and 6. With respect to path traversal, there are at least two opportunities for a CHECK-BEFORE: checking "ftype" before calling strcat() on line 4, or checking "fname" before the fopen() on line 6.

After line 6, there is no CHECK-AFTER to ensure that the file was successfully opened, so there could be a NULL pointer dereference (CWE-476) or other undefined behavior in line 7.

In line 7, the conditional is effectively a CHECK-AFTER that sees if the fgets() call succeeded; if the call failed, then the HANDLE-AFTER is to exit the loop.

In line 8, the code performs a CHECK-BEFORE to filter out dangerous HTML tags to avoid cross-site scripting (CWE-79). However, this check is incomplete, to XSS is still possible. If proper output encoding were used before the printf() in line 9, when this would serve as a HANDLE-BEFORE.

Resource Lifecycle

Resources often have explicit instructions on how to be created, used and destroyed. When software fails to follow these instructions, it can lead to unexpected behaviors and potentially exploitable states.

Within the CWE Research View, the pillar for resource management is:

CWE-664 Improper Control of a Resource Through its Lifetime

Resources have three primary stages:

- INITIALIZATION: the base elements of the resource are allocated and assigned certain properties. Until the initialization is complete, the resource is not intended to be available for use by the application. Otherwise, various weaknesses may occur.

  This is covered by:

  CWE-665 Improper Initialization

  Children of this weakness include Allocation of Resources Without Limits or Throttling (CWE-770) and Missing Initialization (CWE-456).

- USAGE: the resource may be read, modified, copied, or otherwise manipulated. This phase is where the bulk of software activities occur, and where most weaknesses lie.

- RELEASE: the resource is no longer available for use. Any relevant behaviors signal the release, such as by closing a connection by sending a FIN packet. Other actions may be performed, such as releasing lower-layer resources if the original resource is a complex structure or object. In many cases, the resource may be RENEWABLE, i.e., it can be reused by other processes or actors once it has been released. For example, a free() call makes the associated memory available to other functionality in the program, or possibly even to other processes.

  This is covered by:

  CWE-404 Improper Resource Shutdown or Release

  Children of this weakness include memory leaks (CWE-401), finalize() Method Without super.finalize() (CWE-568), and Improper Check for Certificate Revocation (CWE-299). CWE-299 is a child because certificate revocation is effectively a statement that the resource has been released, and CWE-299 is the failure to detect if an externally-influenced resource has been released by the external party.

Throughout this lifecycle, resources are accessed or referred to using IDENTIFIERS or HANDLES. The process of RESOLUTION converts a resource identifier to a single, canonical form. For example, code that converts "/tmp/abc/../def.xyz" to "/tmp/def.xyz" is performing resolution on an identifier that is being used for a file resource.

Weaknesses can arise when an attacker can control an identifier, handle, or reference that can resolve to a resource that is outside of the intended control sphere.

Weaknesses that are related to improper resolution are covered in Use of Incorrectly-Resolved Name or Reference (CWE-706).

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

For the handling of the messages file, there are two separate weaknesses. The initialization step includes the construction of the filename in "fname" (lines 3 through 5), and the opening of a file handle (line 6). At this point, the Bug Barrel does not check to see if the initialization failed - i.e., that the fopen() call in line 6 returned a non-NULL value. As a result, a NULL pointer dereference (CWE-476) can occur in line 7.

Assuming that the fopen() succeeds, the code sends the contents of the file in the while loop in lines 7 and 9. However, it does not release the handle before it returns in line 10, so there is a file descriptor leak (CWE-775).

There is a chance for another initialization error, although it is not clear at this layer. Specifically, if the behavior of Get_Query_Param() is to provide a NULL pointer when the desired parameter is not specified in the URL, then the "ftype" variable will be initialized to an unexpected value, and a NULL pointer dereference occurs in line 4.

As already mentioned, many weaknesses can occur during the usage stage as well. In this case, there is a path traversal weakness (CWE-22) from lines 2, 4, and 6, in which a ".." sequence in ftype will cause the program to read a file that is not in the intended control sphere of the user.

Message Structure between Control Spheres

When communication occurs between control spheres, this communication typically takes the form of structured messages or data. The structured message contains one or more directives, separated by "special elements" and metadata that act as markers for the structure of the message.

In many protocols and specifications, the directive and its supporting data are transferred as a single mixed "stream" within a single channel. Injection-related weaknesses (CWE-74) occur when the product allows an attacker to modify the structure of a message as it is sent across this single stream. A data/directive boundary error occurs when data contains special elements that cause portions to be inadvertently interpreted as directives, or vice versa.

The primary CWE entry for this type of problem is Improper Enforcement of Message or Data Structure (CWE-707). When an incoming message is not properly structured, then the product may behave incorrectly. If the product sends an outgoing message that is not properly structured, then its downstream component may behave incorrectly. This weakness typically applies in cases where the product prepares a control message that another process must act on, such as a command or query, and malicious input that was intended as data, can enter the control plane instead. However, this weakness also applies to more general cases where there are not always control implications.

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

For cross-site scripting (CWE-79), suppose that "fname" is only expected to be a simplified data file containing multiple pre-formatted HTML messages. It is assumed that the file only contains the <b>, <u>, and <i> HTML tags. Any other tag would violate the intended structure of the message file. If the program has allowed attackers to create their own messages without doing any proper validation or encoding, then a variety of strings could modify the intended structure. Even a tag such as "<sCrIpT>" would not be caught by the case-sensitive comparison in line 8. As a result, at line 9, the program can include unexpected scripting syntax in its output to the web browser, modifying the intended structure of the resulting web page.

Simplified Model of Access Control

Access control is a commonly-used protection mechanism, but a wide variety of mechanisms exist. A simplified model may assist in describing mechanism-independent failures.

- PERMISSIONS: the explicit specifications for a resource, or a set of resources, that defines which actors are allowed to access that resource, and which actions may be performed by those actors. Permissions can contribute to the definition of one or more intended control spheres.
- PRIVILEGES: the explicit specifications for an actor that defines which behaviors are allowed by the actor.
- AUTHENTICATION: the process of verifying that an actor has a specific real-world identity, typically by checking for information that the software assumes can only be produced by that actor. This is different than authorization, because authentication focuses on verifying the identity of the actor, not what resources the actor can access.
- AUTHORIZATION: the process of determining whether an actor has the required privileges to access a resource based on the permissions associated with that resource, in conjunction with the implicit and explicit security policies for the system. This is different than authentication, because authorization focuses on whether a given actor can access a given resource, not in proving what the real-world identity of the actor is.
- ACCESS CONTROL: a protection mechanism that performs AUTHENTICATION and AUTHORIZATION to ensure that each actor can only access resources and behaviors within that actor's intended control sphere.

Manipulation Types

Types of Manipulations

There are three main manipulation types:

- REACHABILITY: required to reach the relevant behavior. Example - when a buffer overflow can only occur in the password field, the reachability manipulation involves providing a login name first.
- TRIGGER: modifies the behavior. Examples: long argument, flood of requests. The trigger manipulation usually violates the expected properties of the input.
- FACILITATOR: improves control of behaviors or overcomes limitations imposed by product behaviors. Examples: using alphanumeric shellcode to satisfy filtering requirements, "%00" in Perl/PHP filenames to expand the scope of directory traversal to arbitrary file extensions, or the ">" in the beginning of XSS manipulation that closes off the opening tag that the product has already produced in the output. A SYNTACTIC REALIGNMENT is a facilitator manipulation that allows execution to continue cleanly after the payload has been executed.

Manipulations can be characterized in terms of properties. They can be composed or chained.

Examples of Manipulations and Properties

- Using "../../etc/passwd" might be equivalent to "/etc/passwd" for file operations in which the current working directory is two levels below the / directory.
- If "/a/b/c" filename is a symbolic link to "/etc/passwd," then /a/b/c is equivalent to /etc/passwd in most file operations.
- A binary file and its base64-encoded version are semantically equivalent, assuming the encoding is valid.
- A PHP application vulnerability involving register_globals might violate a property that a variable's value should not be mutable by the product's users.
- A "GET /" without the version is syntactically invalid.
- A stack-based buffer overflow involving a long input string might be syntactically and semantically valid for the protocol's specification, but it might be semantically invalid (too large) for the product's intended policy. When the stack-based overflow occurs, this modifies adjacent variables and violates the intended non-mutability of the stack. When the shellcode is actually executed after the overflow has occurred, that same input is semantically invalid at the product's level, but semantically valid at the OS level, since it's well-formed.
- Session fixation attacks introduce non-mutability when mutability is required.
- Access control can sometimes be bypassed by changing lowercase to uppercase, preserving equivalence.
- In FTP, doing a "LIST" before a "USER/PASS" is semantically invalid.
- Making 100 connections to a server might be semantically valid, although the underlying array that tracks the connections might become "syntactically" invalid.

Directive manipulations might include:

- Skip first step
- Skip required step
- Perform steps out of order
- Perform repeated steps
- Do not finish step

- Interrupt step

Notice how most directive manipulations have equivalent data-driven manipulations. For example, HTTP Parameter Pollution (HPP) involves repeated use of the same parameter.

For attacks that are intended to bypass protection mechanisms, some manipulations include:

- Use equivalence to access a desired object that is being protected by its name
- Use invalid step sequences to directly access a resource instead of going through expected steps
- Access alternate channel, which is assumed to be trusted

Artifact Labels

Artifact Labels are used to identify important locations in code, design, or an algorithm that are relevant to a potential weakness or vulnerability. Vulnerability researchers frequently highlight these locations when presenting vulnerable code, but they do not use the same terminology, if at all. These labels can be useful in describing certain vulnerability topologies in the abstract sense, as well.

- INTERACTION POINT: the location where "input" (of either data or directives) enters the product from an external environment.
- CROSSOVER POINT: the location after which an expected property is violated. This is likely to lead to incorrect actions at a later point. For example, a programmer might use a regular expression to restrict an input string to contain only digits, such as for a telephone number. After applying the regular expression, the string is expected to have the property "only contains digits." If the regular expression is incorrectly specified (e.g. only testing for the presence of a digit anywhere in the string), then after its application, the code reaches a crossover point because the string does not necessarily have the property of "only contains digits."
- TRIGGER POINT: the location in the software after which it can no longer prevent itself from violating the intended security policy without relying on implicit mechanisms at another layer. For example, in buffer overflows, the trigger point occurs when the software writes to a memory location outside of the targeted buffer. With OS command injection, the trigger point occurs when the command is passed from the application to the OS.
- ACTIVATION POINT: the location at which the software violates the intended security policy, i.e., where the attacker's "payload" becomes active; presumably, the payload involve the incorrect behaviors. For example, in SQL injection, the activation point occurs when the database server executes the attacker-modified SQL query.
- ATTACK VECTOR: a tuple of (INTERACTION POINT, CROSSOVER POINT, TRIGGER POINT, ACTIVATION POINT). Different vulnerabilities could have the same attack vector. Different attacks could have different trigger and activation points; for example, a buffer overflow intended for denial of service would have a different activation point than one intended for code execution.

The crossover, trigger, and activation points can appear in different locations - different functions, components, or processes. For example, in HTML injection, the trigger point might be the introduction of XSS sequences into a web page that is generated; however, the payload is not activated until an outside party visits the page with a web browser that has scripting enabled.

Crossover, trigger, and activation points can also be very close together, which typically

occurs with buffer overflows and OS command injection (CWE-78).

Note that there might be additional points of interest, especially at a low level. Also, while this model is suitable for input-related weaknesses, it might not be sufficient for behavior-based weaknesses, such as Incorrect Behavior Order (CWE-696) and Incorrect Control Flow Scoping (CWE-705).

Bug Barrel Example

Consider the Bug Barrel code:

```
1   printf("<title>Blissfully Ignorant, Inc.</title>");
2   ftype =  Get_Query_Param("MessageType");
3   strcpy(fname, "/home/cwe/");
4   strcat(fname, ftype);
5   strcat(fname, ".dat");
6   handle = fopen(fname, "r");
7   while(fgets(line, 512, handle)) {
8     if (strncmp(line,"<script>",8)) {
9        printf(line); } }
10  return(200);
```

For a buffer overflow (CWE-119), the interaction point is the assignment to ftype at line 2. Since there is no check to see if ftype is larger than fname, then the crossover point is between lines 2 and 4 - ftype has a "size" property that is larger than expected by the programmer. The trigger point somewhere within the execution of strcat() on line 4, since strcat() does not prevent overflow. If the attacker has a goal of crashing the software, then the strcat() call may also be the activation point. However, if the attacker's goal is to execute code, then the code is not executed until the function returns at line 10 (assuming fname is allocated on the stack and there are no external protection mechanisms).

For a path traversal (CWE-22) attack, the interaction point is the assignment to ftype at line 2. The crossover is at line 4, since it is clear from line 3 that the programmer assumes that ftype does not have ".." sequences in it. The trigger point is at line 6, because the program has opened an unexpected file for reading. The activation point does not occur until line 9, when the software sends the contents of the unexpected file to the attacker. Note that if the fopen() in line 6 was for a write instead of a read, then the activation point would have occurred at line 6.

For cross-site scripting (CWE-79), the primary interaction point is at line 7, where the input is read from the opened file. Assuming that this file contains HTML tags such as "<sCrIpT>" or tags such as IMG that accept javascript in attributes, then the crossover point happens after line 8. This is because the check for "<script>" tags is an incomplete blacklist (CWE-184), so the line variable in line 9 violates the expected property "contains no script." The trigger point happens in line 9, when the XSS is sent as output to the target browser. Note that the activation point is not reached until the attacker's script is executed within the user's web browser. This demonstrates how artifact labels can cross multiple components.

Note that a more complicated attack may exist. Using path traversal, the attacker might be able to use type-I XSS (or CSRF) to convince a user to make a request with MessageType containing ".." sequences. These in turn might be used with %0 (null bytes) to access a web server log that contains XSS in it. In this case, there are effectively three interaction points - lines 2 and 7, in addition to whatever attack inserted the XSS sequences into the target file in the first place.

Differences between crossover and trigger points

It can be difficult to identify the differences between crossover and trigger points. They often appear at the same line of code in one layer, but they may be distinct at a lower layer.

Much of the challenge occurs when there is a missing protection mechanism, because there might not be any code that can be unambiguously "blamed" for causing the problem. Thus the location of the crossover point can be uncertain. For example, if a generated filename is never checked for directory traversal sequences, then the crossover point could be where the filename is constructed (although it could be checked after construction), or just before a file operation (e.g., delete) is performed on the supplied filename.

The evolving convention has been to identify the crossover point as the last possible place in which an explicit check/handle for the target property can be performed. However, this may be subject to change.

Consider the following code:

```
LongString = GetUntrustedInput();
printf("Hello world!");
strcat(x, LongString);   /* overflow */
```

The crossover can be identified as occurring in the function call to strcat() - at this layer, there is no opportunity to enforce the expected properties of LongString. At a lower layer, within the implementation of the strcat(), the trigger point effectively occurs somewhere within a memory copy operation.

Notes on Terminology

For interaction points, "injection" might be a more natural term. However, the term is already overloaded within various communities, and current usage is data-centric. Interaction points can be equivalent to what others call "entry points," but that term has different uses in binary reverse engineering.

Artifact Labels and Protection Mechanisms

From an application perspective, different protection mechanisms may be useful at different stages of an attack. These can be described in association with artifact labels. A defense-in-depth strategy may include protection mechanisms at each stage.

- Before Interaction Point: an application firewall may sanitize inputs before the input is even provided to the application. This is outside the control of the application itself.
- Between Interaction and Crossover: Explicit mechanisms such as input validation attempt to prevent resources from having unexpected properties. These are under the control of the application.
- Between Crossover and Trigger: it is likely that few mechanisms exist to provide appropriate protection in this phase. This is because, by definition, a resource does not have an assumed property after the crossover point has been reached. There may be some lower-layer protections available, such as an input validation step within an API function.
- Between Trigger and Activation: at this point, there may be a reliance on externally-provided mechanisms such as canary-based stack overflow protection. In multi-stage injection attacks (e.g. second-order SQL injection), a protection mechanism on the downstream component may be available. For example, an attacker might be able to reach a trigger point in which untrusted XSS or SQL syntax is inserted into a database, but if the downstream component performs proper input validation and output encoding when it reads that data from the database, the activation point is avoided.
- After Activation: by definition, once the activation point has been reached, the attacker has already violated the software's intended control sphere. At this stage, only external mechanisms can limit the damage, such as role-based access control, privilege separation or jails, or automated intrusion detection and prevention, etc.

Additional Concepts under Exploration

The following concepts are probably important, but they haven't been explored as fully as the others.

- ASSUMPTIONS/EXPECTATIONS: The assumptions that a program, design, or API makes regarding the properties of behaviors and resources. The human developer makes certain assumptions; when code executes, it has certain expectations about its environment, data, etc., and how its behaviors have modified program state.
- DESIGN LIMITATION: a feature or behavior that can theoretically be used correctly, but could lead to a vulnerability if not. For example, the functionality of strcpy() is a design limitation. It can be used securely, but it can introduce vulnerabilities if used incorrectly. This incorrect usage would be an implementation flaw (and perhaps, usage of strcpy at all might be regarded as a design flaw in some circles). Conjecture: all implementation bugs are associated with at least one design limitation.
- DEGREES OF CONTROL: how much control an actor has over a resource; this is frequently described in terms of attacker control over data, directives, and consequences. An actor might have Full, Partial, or No control over a resource - and this could change over time.
- PRINCIPLES: everyone's definition of "vulnerability" differs, but it could be defined in terms of certain principles, such as "users should not have access to any resource that is not explicitly granted or implied." Defining these principles could become the basis of a Universal Policy.
- BOUNDARIES/INTERFACES: the boundaries between multiple actors, components, etc. Many (but probably not all) issues occur at boundaries or interfaces between two different entities. Boundaries *might* include representation, data, process/module, actor, etc. Representation is probably essential for adequately explaining major vulnerability phyla such as "injection".
- OUTPUTS: Wing et al. explicitly model "exit points" as places where data exits a system; web application security people talk a lot about "output validation." This notion is useful when examining a system/actor in isolation, but a framework that can cover all aspects of attacks/vulnerabilities might only need to model "input." Information leaks can be thought of as output, but it's only a leak if it's an input to an attacker.

- CONTAINERS/SANDBOXES: these seem to apply mostly to files and code, but thinking of vulnerabilities as they relate to containers has sometimes been useful. For example, directory traversal and some Java sandbox escaping works by using syntactically valid manipulations that produce references to resources outside the container, which are semantically invalid relative to the intended policy. PHP file inclusion can be thought of as a violation of an intended container that uses semantically invalid manipulations.

- "EXPLICIT" vs. "EMERGENT" - behaviors, properties, and resources can either be explicit or emergent. For example, a covert channel can be an emergent resource that wasn't originally intended.

The relationship between design limitations and implementation errors needs more study. Protection mechanisms seem to have unique characteristics in comparison to typical vulnerabilities, and it might be useful to distinguish between "missing" versus "incorrect" protection mechanisms.

Examples of the Terminology in Action

The following items don't contain any revelations that would be surprising for expert researchers. The point is to demonstrate the efficiency of the vocabulary.

- 1) Any protection mechanism that relies on names or identifiers should defend against equivalence manipulations. A product, language, or environment should attempt to minimize the number and types of equivalence manipulations, which are common factors in cross-site scripting and path traversal.

- 2) Fuzzers are very good at finding resultant weaknesses and consequences, but unless the fuzzing is structured, there is no indication of the primary weakness. Diagnosis can be made more difficult if the trigger and activation points are not close by. Unstructured fuzzing will often fail due to the lack of reachability manipulations, e.g. if a parser requires certain semantic consistency between data elements, before a vulnerable behavior can even be invoked.

- 3) Black box testing is likely to fail if the techniques do not consider whether an activation point might be in another process or channel.

- 4) Monitors and Intermediaries are especially subject to equivalence and validity manipulations, since the receiving/target hosts might have alternate interpretations. Example: a web app firewall might allow invalid HTML through, even though the victim's browser converts that HTML into "valid" HTML. An intermediary might choose to act on header X, when its semantically equivalent header Y is what's actually processed by the client.

- 5) When protection mechanisms are involved, a manipulation might originally be syntactically invalid before the mechanism, but then syntactically/semantically valid after the mechanism Example: "..../" in directory traversal is syntactically invalid, until a bad filtering scheme collapses the string into "../". Double-decoding issues are similar.

- 6) XSS and buffer overflows can share certain characteristics, such as the mixture of data and directives. In the overflow case, though, this mixture occurs at a level below the programming language; for XSS, it's at a level above the programming language.

Example Applications of the Theory

There are several potential applications, only two of which are covered here.

- 1) Gap analysis and finding new vulnerability classes

  By moving up a level of abstraction from classes like buffer overflows, XSS, and privilege management errors, we might be able to use the framework to describe new issues in vulnerability theory terms, then look at other known instances that share similar characteristics.

This would help identify gaps in understanding (or current researcher focus), and possibly lead to discovery of new vulnerability classes, or at least variants. Example: Product class X has behaviors B1 and B2, with manipulations M1...M5 on resource R. These manipulations preserve property P and modify property Q. What types of vulnerabilities or attacks involve P and Q, and therefore might be able to affect X?

- 2) Evaluating vulnerability "difficulty"

  Since we expect products to always have vulnerabilities, we hope that they only have the most difficult-to-find, difficult-to-exploit vulnerabilities. Concepts such as artifact labels could be used to calculate the "distance" from input to exploit; novelty and complexity of manipulations could be evaluated more cleanly; actors and channels could be used to describe a "topology" ("vulnerability surface"); and protection mechanisms could be assessed in terms of the properties they preserve.

  For example, if a vulnerability has the interaction, trigger, and activation points all in the same function, that's probably a more obvious vulnerability than something that involves multiple actors, channels, and manipulations.

Future Work

There are several opportunities for future work in this area.

- 1) There will continue to be a synergistic relationship with CWE development and maintenance, especially with respect to the Research View (CWE-1000).
- 2) Relationships with CAPEC will be investigated more closely, especially with respect to concepts such as manipulations, properties, and an attacker's intended policy.
- 3) Further clarification of layers and perspectives
- 4) Clarify relationships with the CWE formalization project led by KDM Analytics. Many CWE weaknesses lack the needed precision, but ideally, vulnerability theory should be able to support machine-findable constructs as identified in white box definitions, although this may happen at a different layer than the current focus.
- 5) Extend beyond the data-driven focus. There has been some effort to ensure that directives also have sufficient coverage for related concepts such as properties, but gaps are likely, such as temporal consistency and atomicity.
- 6) The work in security policies, control spheres, and protection mechanisms may provide a mechanism for formulating "micro-policies" at an algorithmic level, instead of a product level, which has been the emphasis of previous academic research. More work may need to be done with properties to support this.

Related Work

Dowd et al's "Art of Software Security Assessment" touches on some of these concepts in an introductory chapter, but it does not propose them as a formal framework. Our work is more detailed in this respect.

The work from Jeannette Wing et al on measuring attack surface introduces some concepts that overlap vulnerability theory, but it is largely for data-driven attack vectors and is focused on quantitative measurements of design quality.

The Trike threat modeling framework has similar concepts.

Informal conversations with Matt Bishop of UC Davis suggest some overlap with their current work.

One early reviewer suggested that dataflow diagramming has some utility, and using that terminology where appropriate might be useful in educating non-security practitioners.

As of July 2009, the most novel elements of vulnerability theory include behaviors, properties, and control spheres.

Credits

Bill Heinbockel contributed to the 2007 version of this document.

Chris Wysopal, David Litchfield, and Ivan Arce all provided commentary on pre-2007 versions of this document.

Janis Kenderdine advocated the theory that path traversal issues are composites.

Tom Stracener provided supportive feedback on the 2007 version.

Changelog

1.0.1 - October 29, 2009

- changed Error Handling Life Cycle section to a sub-section
- no other changes were performed

1.0 - July 27, 2009

- added control spheres
- added error handling model
- added access control model
- added protection mechanisms
- provided additional details for properties, actors, etc.
- created new "Bug Barrel" and introduced into sections

0.5 - July 9, 2007

- extended definitions
- more examples of specific concepts
- prepared for public release

0.4 - April 7, 2007

- reasons lost

0.3 - Feb 26, 2007

- added crossover points, related work

0.2 - Feb 14, 2007

- added minor points based on feedback

0.1 - Jan 31, 2007

- first version
