"Moving to the Left": Getting Ahead of Vulnerabilities by Focusing on Weaknesses

Jim Duncan, Juniper Networks SDL Program
Agenda

• Problem Statement
• Vulnerabilities vs. Weaknesses
• The Common Weakness Enumeration (CWE)
• Example CWE Entry: CWE-119
• Weaknesses as Bug-Tracking Data
• Tips’n’Tricks for assigning CWE labels
• Futures, Prognostications, Recommendations
• Questions?
Juniper’s Secure Development Lifecycle

• Mission: Improve the resilience of Juniper products through the application of distinct practices within Juniper’s existing engineering processes
• Intended to be lightweight and minimally disruptive
  • “Self-inserting” with low impact
• Broad mission across all Juniper development
  • Program formally started in 2013
• The SDL is not simply desirable
  • Required by a growing number of our customers
  • An obligation on behalf of the Internet community
• SDL Functional Goal: **Vulnerability Containment**
  • Keep product security vulnerabilities inside Juniper
  • Vulnerabilities are much more expensive after they escape
Problem Statement
Vulnerability Chasing is “Whack-a-Mole”

• This is a common thread; that probably means something important.
  • (By the way, those are gophers, not moles. But I digress...)
Hazardous to IR Teams and Our Community

• No surprise that endless reactive work wears us down
• Causes long-term damage
  • Staff health, relationships
  • Team dynamics
  • Team-to-team relationships
  • Personal burn-out
  • Career burn-out
  • Industry burn-out?
• Which did you choose?

“SURVEY SAYS...”

VULNERABILITY HANDLING IS:

- STRESSFUL: 14
- EXPENSIVE: 8
- EXHAUSTING: 4
- FRUSTRATING: 3

VULNERABILITY HANDLING IS:

STRESSFUL EXPENSIVE EXHAUSTING FRUSTRATING

- £ 
- £
- £
- £
Weakness Mitigation is Proactive

"Work smarter, not harder."

• Weaknesses are the constituent elements of vulnerabilities.
  • Weakness resolution eliminates multiple vulnerabilities.
  • Resolution of weaknesses contributes to persistent resilience.
• Unfortunately, it brings its own challenges to the effort:
  • No immediately obvious result; difficult to measure effectiveness.
  • Pays no direct dividend; difficult to justify resources.
  • Immature area of study; nascent topic, much remains to be figured out.
• More to follow on all these points...
Vulnerabilities vs. Weaknesses
Essential Terminology & Key Concepts

• A *vulnerability* depends upon one or more *weaknesses*.
  • In other words, *weaknesses* are the constituent elements of *vulnerabilities*.

• The existence of a *weakness* does not constitute a *vulnerability*.
  • A weakness may be provably identified in some code, but it may not be vulnerable for various reasons. (e.g., unreachable instruction)

• A vulnerability is *actionable*; it violates a security policy.

• But a weakness only has the *potential* to violate a security policy.

• There are more, but these are essential

• Any questions before we continue?
Implications and Misunderstandings

• Weaknesses are an abstraction; they are not well understood.
• Multiple weaknesses may be identified in one snippet of code.
  • Various disparate weakness labels may all be correct; never exclusive.
  • Many individual weaknesses can be argued to be the best possible label.
• Weakness is similar, but not identical, to the concept of root cause.
  • This conflict causes enormous confusion (and occasional conflict)
    • Root cause is generally unitary, atomic, exclusive.
    • Weaknesses are multiplex.
• Code runs fine with weaknesses, but fails with a vulnerability.
• Not confined to code; weaknesses happen in design and elsewhere.
The Common Weakness Enumeration (CWE)
Virtual smörgåsbord of weakness labels, both discrete and abstract.

- Discrete labels, e.g.: **CWE-193: Off-by-one Error** or **CWE-369: Divide by Zero**
- Both are members of **CWE-682: Incorrect Calculation**
- CWE-682 is a member of **CWE-189: Numeric Errors**
- CWE-189 is a member of **CWE-699: Development Concepts**

- These parent-child relationships are not exclusive
  - It’s more like “It takes a village...”: aunts, uncles, grandparents, neighbors!
- Which one do you use? **It depends on the goal you have in mind.**
Weakness Classification and Heirarchy

• *Weakness Base* is the fundamental entity in the compendium.
  • CWE-552: Files or Directories Accessible to External Parties
• *Weakness Variant* is more specific.
  • CWE-553: Command Shell in Externally Accessible Directory
• *Weakness Class* is a higher-level abstraction.
  • CWE-668: Exposure of Resource to Wrong Sphere
• *Category* is a broad set of weaknesses with a common characteristic.
  • CWE-361: 7PK – Time and State
  • “7PK” is *The Seven Pernicious Kingdoms*, a foundational reference for CWE.
Additional Weakness Groupings & Terms

• **Compound Elements**: Closely associated independent weaknesses
  • Based on interaction or co-occurrence, e.g., *Composite* or *Chain* weaknesses

• **Composite**: Simultaneous weaknesses that create a vulnerability.
  • CWE-352: Cross-Site Request Forgery (CSRF)

• **Loose Composite**: Appears to be individual, but isn’t. (Ignore for now.)

• **Chain**: Serialized weaknesses that create a vulnerability.

• **Named Chain**: A chain so frequent that it deserves its own moniker.
  • CWE-692: Incomplete Blacklist to Cross-Site Scripting
And More Weakness Groupings & Terms

• **View**: A useful way of considering CWE content, e.g., a *Slice* or *Graph*
  
• **Implicit Slice**: membership based on common shared characteristic:
  • CWE-919: Weaknesses in Mobile Applications
  • CWE-701: Weaknesses Introduced During Design

• **Explicit Slice**: membership based on some external criterion:
  • CWE-604: Deprecated Entries
  • CWE-630: DEPRECATED: Weaknesses Examined by SAMATE

• **Graph**: membership based on hierarchical relationships:
  • CWE-1026: Weaknesses in OWASP Top Ten (2017)
  • CWE-868: Weaknesses Addressed by the CERT C++ Secure Coding Standard
Confusing Adjectives: Primary v. Resultant

• Primary Weakness: Initial critical error (possibly a root cause) that exposes subsequent weaknesses after it.

• Resultant Weakness: Exposed following an earlier weakness.

• Example:
  • CWE-190: Integer Overflow may cause a size error allocating a buffer.
  • CWE-120: Buffer Overflow occurs because of the preceding size error.

• This relationship is described as
  • ”CWE-190 is primary to CWE-120,” or
  • “CWE-120 is resultant from CWE-190”.
What’s The Purpose of All This?

• For immediate resolution/mitigation, focus on *Base Weaknesses*.
• Nitpick with *Variants* if necessary, maybe *Chains* and *Composites*.
• For long-term improvement/refinement, look at *Class* and *Category*.
• Constrain with *Views*, especially *Slices*. **Important**! (more on this later)

• What does all this look like in practice?
Example

CWE Entry:
CWE-119
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Status: Usable

Description
The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

Extended Description
Certain languages allow direct addressing of memory locations and do not automatically ensure that these locations are valid for the memory buffer that is being referenced. This can cause read or write operations to be performed on memory locations that may be associated with other variables, data structures, or internal program data.
As a result, an attacker may be able to execute arbitrary code, alter the intended control flow, read sensitive information, or cause the system to crash.

Alternate Terms

Memory Corruption: The generic term "memory corruption" is often used to describe the consequences of writing to memory outside the bounds of a buffer, when the root cause is something other than a sequential copies of excessive data from a fixed starting location (i.e., classic buffer overflows or CWE-120). This may include issues such as incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release, etc.

Relationships
The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOf and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that the user may want to explore.

Relevant to the view "Research Concepts" (CWE-1000)
Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)
Relevant to the view "Development Concepts" (CWE-699)
Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)
<table>
<thead>
<tr>
<th>Nature</th>
<th>Type</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChildOf</td>
<td></td>
<td>118</td>
<td>Incorrect Access of Indexable Resource ('Range Error')</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>123</td>
<td>Write-what-where Condition</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>125</td>
<td>Out-of-bounds Read</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>466</td>
<td>Return of Pointer Value Outside of Expected Range</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>680</td>
<td>Integer Overflow to Buffer Overflow</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>786</td>
<td>Access of Memory Location Before Start of Buffer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>787</td>
<td>Out-of-bounds Write</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>788</td>
<td>Access of Memory Location After End of Buffer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>805</td>
<td>Buffer Access with Incorrect Length Value</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>822</td>
<td>Untrusted Pointer Dereference</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>823</td>
<td>Use of Out-of-range Pointer Offset</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>824</td>
<td>Access of Uninitialized Pointer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>825</td>
<td>Expired Pointer Dereference</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>20</td>
<td>Improper Input Validation</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>128</td>
<td>Wrap-around Error</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>129</td>
<td>Improper Validation of Array Index</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>131</td>
<td>Incorrect Calculation of Buffer Size</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>190</td>
<td>Integer Overflow or Wraparound</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>193</td>
<td>Off-by-one Error</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>195</td>
<td>Signed to Unsigned Conversion Error</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>839</td>
<td>Numeric Range Comparison Without Minimum Check</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>843</td>
<td>Access of Resource Using Incompatible Type ('Type Confusion')</td>
</tr>
</tbody>
</table>
### Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

<table>
<thead>
<tr>
<th>Nature</th>
<th>Type</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChildOf</td>
<td></td>
<td>118</td>
<td>Incorrect Access of Indexable Resource (‘Range Error’)</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>123</td>
<td>Write-what-where Condition</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>125</td>
<td>Out-of-bounds Read</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>787</td>
<td>Out-of-bounds Write</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>824</td>
<td>Access of Uninitialized Pointer</td>
</tr>
</tbody>
</table>

### Relevant to the view "Development Concepts" (CWE-699)

<table>
<thead>
<tr>
<th>Nature</th>
<th>Type</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemberOf</td>
<td></td>
<td>19</td>
<td>Data Processing Errors</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>120</td>
<td>Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>123</td>
<td>Write-what-where Condition</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>125</td>
<td>Out-of-bounds Read</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>130</td>
<td>Improper Handling of Length Parameter Inconsistency</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>786</td>
<td>Access of Memory Location Before Start of Buffer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>787</td>
<td>Out-of-bounds Write</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>788</td>
<td>Access of Memory Location After End of Buffer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>805</td>
<td>Buffer Access with Incorrect Length Value</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>822</td>
<td>Untrusted Pointer Dereference</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>823</td>
<td>Use of Out-of-range Pointer Offset</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>824</td>
<td>Access of Uninitialized Pointer</td>
</tr>
<tr>
<td>ParentOf</td>
<td></td>
<td>825</td>
<td>Expired Pointer Dereference</td>
</tr>
<tr>
<td>CanFollow</td>
<td></td>
<td>131</td>
<td>Incorrect Calculation of Buffer Size</td>
</tr>
</tbody>
</table>

### Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

<table>
<thead>
<tr>
<th>Nature</th>
<th>Type</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChildOf</td>
<td></td>
<td>20</td>
<td>Improper Input Validation</td>
</tr>
</tbody>
</table>
### Modes Of Introduction

The different Modes of Introduction provide information about how and when this weakness may be introduced. The Phase identifies a point in the software life cycle at which introduction may occur, while the Note provides a typical scenario related to introduction during the given phase.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture and Design</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td></td>
</tr>
</tbody>
</table>

### Applicable Platforms

The listings below show possible areas for which the given weakness could appear. These may be for specific named Languages, Operating Systems, Architectures, Paradigms, Technologies, or a class of such platforms. The platform is listed along with how frequently the given weakness appears for that instance.

**Languages**
- C (Often Prevalent)
- C++ (Often Prevalent)
- Class: Assembly (Undetermined Prevalence)

### Common Consequences

The table below specifies different individual consequences associated with the weakness. The Scope identifies the application security area that is violated, while the Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness. The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Impact</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>Technical Impact: Execute Unauthorized Code or Commands; Modify Memory</td>
<td></td>
</tr>
<tr>
<td>Confidentiality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>Technical Impact: Read Memory; DoS: Crash, Exit, or Restart; DoS: Resource Consumption (CPU); DoS: Resource Consumption (Memory)</td>
<td></td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.</td>
<td></td>
</tr>
<tr>
<td>confidentiality</td>
<td>Technical Impact: Read Memory</td>
<td></td>
</tr>
<tr>
<td>confidentiality</td>
<td>In the case of an out-of-bounds read, the attacker may have access to sensitive information. If the sensitive information contains system details, such as the current buffers position in memory, this knowledge can be used to craft further attacks, possibly with more severe consequences.</td>
<td></td>
</tr>
</tbody>
</table>

### Likelihood Of Exploit

**High**

---

CWE-119, 4/8

Duncan, "Moving to the Left": Getting Ahead of Vulnerabilities by Focusing on Weaknesses
Demonstrative Examples

Example 1
This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: C

```c
void host_lookup(char *user_supplied_addr){
    struct hostent *hp;
    in_addr_t *addr;
    char hostname[64];
    in_addr_t inet_addr(const char *cp);

    /*routine that ensures user_supplied_addr is in the right format for conversion*/
    validate_addr_form(user_supplied_addr);
    addr = inet_addr(user_supplied_addr);
    hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
    strcpy(hostname, hp->h_name);
}
```

This function allocates a buffer of 64 bytes to store the hostname, however there is no guarantee that the hostname will not be larger than 64 bytes. If an attacker specifies an address which resolves to a very large hostname, then we may overwrite sensitive data or even relinquish control flow to the attacker.

Note that this example also contains an unchecked return value (CWE-222) that can lead to a NULL pointer dereference (CWE-476).

Example 2
This example applies an encoding procedure to an input string and stores it into a buffer.

Example Language: C

```c
char * copy_input(char *user_supplied_string){
    int i, dst_index;
    char *dst_buf = (char*)malloc(sizeof(char) * MAX_SIZE);
    if ( MAX_SIZE <= strlen(user_supplied_string) )
        die("user string too long, die evil hacker!");
    dst_index = 0;
    for ( i = 0; i < strlen(user_supplied_string); i++ )
        if ( &i == user_supplied_string[i] )
            dst_buf[dst_index++] = '\x';
        dst_buf[dst_index++] = a';
        dst_buf[dst_index++] = '\m';
        dst_buf[dst_index++] = p';
        dst_buf[dst_index++] = ';';
    }
```
Example 4

In the following code, the method retrieves a value from an array at a specific array index location that is given as an input parameter to the method.

Example Language: C

```c
int getValueFromArray(int *array, int len, int index) {
    int value;
    // check that the array index is less than the maximum
    // length of the array
    if (index < len) {
        // get the value at the specified index of the array
        value = array[index];
    } // if array index is invalid then output error message
    // and return value indicating error
    else {
        printf("Value is: %d\n", array[index]);
        value = -1;
    }
    return value;
}
```

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in an out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

Example Language: C

```c
...

// check that the array index is within the correct
// range of values for the array
if (index >= 0 && index < len) {
    ...
```
**Observed Examples**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2009-2550</td>
<td>Classic stack-based buffer overflow in media player using a long entry in a playlist</td>
</tr>
<tr>
<td>CVE-2009-2403</td>
<td>Heap-based buffer overflow in media player using a long entry in a playlist</td>
</tr>
<tr>
<td>CVE-2009-0689</td>
<td>Large precision value in a format string triggers overflow</td>
</tr>
<tr>
<td>CVE-2009-0690</td>
<td>Negative offset value leads to out-of-bounds read</td>
</tr>
<tr>
<td>CVE-2009-1532</td>
<td>Malformed inputs cause accesses of uninitialized or previously-deleted objects, leading to memory corruption</td>
</tr>
<tr>
<td>CVE-2009-1528</td>
<td>Chain: lack of synchronization leads to memory corruption</td>
</tr>
<tr>
<td>CVE-2009-0558</td>
<td>Attacker-controlled array index leads to code execution</td>
</tr>
<tr>
<td>CVE-2009-0269</td>
<td>Chain: -1 value from a function call was intended to indicate an error, but is used as an array index instead.</td>
</tr>
<tr>
<td>CVE-2009-0566</td>
<td>Chain: incorrect calculations lead to incorrect pointer dereference and memory corruption</td>
</tr>
<tr>
<td>CVE-2009-1350</td>
<td>Product accepts crafted messages that lead to a dereference of an arbitrary pointer</td>
</tr>
<tr>
<td>CVE-2009-0191</td>
<td>Chain: malformed input causes dereference of uninitialized memory</td>
</tr>
<tr>
<td>CVE-2008-4113</td>
<td>OS kernel trusts userland-supplied length value, allowing reading of sensitive information</td>
</tr>
<tr>
<td>CVE-2003-0542</td>
<td>Buffer overflow involving a regular expression with a large number of captures</td>
</tr>
<tr>
<td>CVE-2017-1000121</td>
<td>Unchecked message size metadata allows integer overflow (CWE-190) leading to buffer overflow (CWE-119).</td>
</tr>
</tbody>
</table>

**Potential Mitigations**

### Phase: Requirements

**Strategy: Language Selection**

Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer.

Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.

### Phase: Architecture and Design

**Strategy: Libraries or Frameworks**

Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Examples include the Safe C String Library (SafeStr) by Messier and Viega [REF-57], and the Strsafe.h library from Microsoft [REF-56]. These libraries provide safer versions of overflow-prone string-handling functions.

Note: This is not a complete solution, since many buffer overflows are not related to strings.

### Phase: Build and Compilation

**Strategy: Compilation or Build Hardening**

Run or compile the software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows.

For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio Fedor/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.
### Taxonomy Mappings

<table>
<thead>
<tr>
<th>Mapped Taxonomy Name</th>
<th>Node ID</th>
<th>Fit</th>
<th>Mapped Node Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWASP Top Ten 2004</td>
<td>A5</td>
<td>Exact</td>
<td>Buffer Overflows</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>ARR00-C</td>
<td></td>
<td>Understand how arrays work</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>ARR30-C</td>
<td>CWE More Abstract</td>
<td>Do not form or use out-of-bounds pointers or array subscripts</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>ARR38-C</td>
<td>CWE More Abstract</td>
<td>Guarantee that library functions do not form invalid pointers</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>ENV01-C</td>
<td>CWE More Abstract</td>
<td>Do not make assumptions about the size of an environment variable</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>EXP39-C</td>
<td>Imprecise</td>
<td>Do not access a variable through a pointer of an incompatible type</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>FI037-C</td>
<td></td>
<td>Guarantee that storage for strings has sufficient space for character data and the null terminator</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>STR31-C</td>
<td>CWE More Abstract</td>
<td>Do not pass a non-null-terminated character sequence to a library function that expects a string</td>
</tr>
<tr>
<td>CERT C Secure Coding</td>
<td>STR32-C</td>
<td>CWE More Abstract</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>WASC</td>
<td>7</td>
<td>SFP8</td>
<td>Faulty Buffer Access</td>
</tr>
</tbody>
</table>

### Related Attack Patterns

- **CAPEC-ID**
  - **CAPEC-10**: Buffer Overflow via Environment Variables
  - **CAPEC-100**: Overflow Buffers
  - **CAPEC-14**: Client-side Injection-induced Buffer Overflow
  - **CAPEC-24**: Filter Failure through Buffer Overflow
  - **CAPEC-42**: MIME Conversion
  - **CAPEC-44**: Overflow Binary Resource File
  - **CAPEC-45**: Buffer Overflow via Symbolic Links
  - **CAPEC-46**: Overflow Variables and Tags
  - **CAPEC-47**: Buffer Overflow via Parameter Expansion
  - **CAPEC-8**: Buffer Overflow in an API Call
  - **CAPEC-9**: Buffer Overflow in Local Command-Line Utilities

### References


Weaknesses as Bug-Tracking Data
Case Study: CWE Implemented in GNATS

• Initial implementation: One drop-down field next to SIRT data.
  • Advantage: match-as-you-type provides rapid lookup and completion.
  • Disadvantages: only one possible label, feeble support for novices.

• Better implementation supports a list of CWE labels with priority.
  • Constrained to one value, lots of time spent on finding best possible CWE.
  • Multiple values provides cache and depth, better holistic understanding.
  • Also would include a grouping function for trending support and refinement.

• Immediate benefit: CWE link helps developer understand flaw.
  • Varies among entries, but many have deep, broad explanations, suggestions.
  • Includes recommendations for recognizing good and bad examples.
Weakness Data as SDL Deliverable

• Roll-up of weakness groups produces trends in pentest findings
  • Exposes areas of good performance and others needing remediation
  • Feeds back into Training practice, Security in Design practice
  • Identifies areas for focus by development teams and leadership

• Annual internal ”Top Ten Weaknesses” Report
  • Long-term trending of bug reports across all products
  • Data is compared and contrasted with industry results (OWASP, SANS/CWE)

• Future goal: automatic reporting of CWEs from static code analysis
  • Caveat: to be tagged; static analyzers produce “stilted” results
Tips’n’Tricks for Assigning CWE Labels
Researcher? Or Developer? Start Here!

• If you are the executive of some organization, you care about **impact**.
  • E.g., denial of service, data exfiltration, log errors causing compliance failures
  • “Research Concepts” provides a useful constraint for filtering selections

• If you are a software/hardware vendor, you care about **cause**.
  • E.g., buffer overflow, trust boundary failure, poor entropy, authorization flaw
  • “Development Concepts” should be your default constraint (but not solid)

• **If I had this to do over again, I would have started with this emphasis.**
Use the Search and Filter Functions Well

• Prefix search terms with “inurl:definitions” to constrain to CWEs only.
  • E.g., “Search: inurl:definitions entropy”
  • Otherwise, you’ll hit matches from all over the CWE web site.

• Select the appropriate ”Presentation Filter” near the top, left side.
  • I almost always use “Complete”, but that may be overwhelming.
  • Other choices are “Basic”, “High Level”, “Mapping-Friendly”.

• Or simply jump directly to the CWE of interest.
  • The jump field is located in the top-right corner of every page.
  • Enter only the numeric part of the CWE identifier.
Can’t Decide on the Best CWE?

• Try widening your search, then narrowing again.
  • Move up to one of the parents, then peruse the children.
  • No result? Move back and try a different parent. “Lather, rinse, repeat.”

• Apply reverse engineering to CWE itself.
  • Browse the CVE and CAPEC links; both have links back into CWE.
  • Warning: NVD/CVE links to CWE are alarmingly general, sometimes wrong!
  • Note that just like CVSS, different vendors correctly have different CWEs.

• Don’t forget the References; sometimes a deeper study is needed.

• Third-party tools, e.g., CWEvis, may be helpful.
  • I have no current experience with CWEvis due to browser issues
Futures, Prognostications, Recommendations
Why Fix Weaknesses? The Code Works Fine!

• Consider this conversation:
  • Development Manager: “How did you spend your day?”
  • Developer: “I found and resolved two weaknesses in our main code.”
  • Manager: “Did you finish your feature requests?”
  • Developer: “No, but I fixed these two weaknesses.”
  • Manager: “Was the code running fine before?”
  • Developer: “Yes, of course, but it was weak. I fixed these weaknesses!”
  • Manager: “!!!!!”....

• Major awareness is needed all across the industry.
Customers/Auditors Shifting Focus to CWE

• Customers (and specifically their auditors) are moving beyond requirements that specific CVEs be fixed; they are now tracking CWEs.
  • Considerable confusion; recall that this is all new to most participants.
  • Note that “CVE/CWE” may be an example of “CWE-193: Off-by-one Error”. ;-)

• This brings up a whole new set of complications.
  • How do we audit this? How do we measure compliance?
  • What about multiple weaknesses? Do we look at chains and composites, too?
  • When does this become legally important, as in “acceptable to the trade”?
What’s Next?

• We should expect/request better metrics, more exploration of trends.
  • PhD dissertation ideas? FIRST Metrics SIG? Other SIGs? Maybe a CWE SIG?
  • Better input/management of label determination, parentage, peers.
  • Awareness needed for weakness science and improvement of terminology.

• CWE needs broader community support and more infrastructure.
  • How do I suggest a new CWE label with sensitivity implications?
  • Broad areas are missing, e.g., IP violations; What else should be addressed?

• More ideas? Email me! jduncan@juniper.net
Questions?

Thank You!