The NIST Bugs Framework (BF)

https://samate.nist.gov/BF/
My Background ➔ Quite Excited about BF

- Ph.D. Dissertation – Static Analysis, Simulation, and Verification of Formal Specifications:
- Fascinated by programming paradigms
- Developed formal specification languages
- BF – Dreams come true
Agenda

- Existing Repositories:
  - CWE
  - CVE
  - NVD
  - KEV

- Example – Heartbleed

- The Bugs Framework (BF)
  - Early Work
  - Terminology
  - Goals
  - Features

- Potential Impacts
Existing Repositories
Commonly Used Repositories

- Weaknesses:
  - **CWE** – Common Weakness Enumeration

- Vulnerabilities:
  - **CVE** – Common Vulnerabilities and Exposures
    → over 18 000 documented in 2020

- Linking weaknesses to vulnerabilities – CWEs to CVEs:
  - **NVD** – National Vulnerabilities Database

- By priority for remediation – CVEs:
  - **KEV** – Known Exploited Vulnerabilities Catalog
Repository Problems

1. Imprecise Descriptions – CWE & CVE
2. Unclear Causality – CWE & CVE
3. No Tracking Methodology – CVE
4. Gaps in Coverage – CWE
5. Overlaps in Coverage – CWE
6. No Tools – CWE & CVE
CWE-502: Deserialization of Untrusted Data:
The application deserializes untrusted data without *sufficiently verifying that* the resulting data *will be valid.*

- Unclear what "*sufficiently*" means,
- "*verifying that data is valid*" is also confusing
Example:

**CVE-2018-5907**
Possible buffer overflow in msm_adsp_stream_callback_put due to lack of input validation of user-provided data that leads to integer overflow in all Android releases (Android for MSM, Firefox OS for MSM, QRD Android) from CAF using the Linux kernel.

→ the NVD label is **CWE-190**

While the CWEs chain is:
CWE-20 → CWE-190 → CWE-119
Problems #4, #5: Gaps/Overlaps in Coverage

- Example:

  CWEs coverage of buffer overflow by:
  - ✓ Read/ Write
  - ✓ Over/ Under
  - ✓ Stack/ Heap

<table>
<thead>
<tr>
<th></th>
<th>Over</th>
<th>Under</th>
<th>Either End</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>CWE-127</td>
<td>CWE-126</td>
<td>CWE-125</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Write</td>
<td>CWE-124</td>
<td>CWE-120</td>
<td>CWE-123</td>
<td>CWE-787</td>
<td>CWE-121</td>
</tr>
<tr>
<td>Read/ Write</td>
<td>CWE-786</td>
<td>CWE-788</td>
<td>★</td>
<td>★</td>
<td>★</td>
</tr>
</tbody>
</table>
The Bugs Framework (BF)
Example:

CVE versus BF

Descriptions of Heartbleed
The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.
Heartbleed (CVE-2014-0160)

The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.

```c
1448 dtls1_process_heartbeat(SSL *s)
1449 {
1450  unsigned char *p = &s->s3->rrec.data[0], *pl;
1451  unsigned short hbtype;
1452  unsigned int payload;
1453  unsigned int padding = 16; /* Use minimum padding */
1454  /* Read type and payload length first */
1455  hbtype = *p++;
1456  n2s(p, payload);
1457  pl = p;
1458  ...
1459  if (hbtype == TLS1_HB_REQUEST)
1460    {
1461      unsigned char *buffer, *bp;
1462      /* Allocate memory for the response, size is 1 byte message type, plus 2 bytes payload, plus payload, plus padding */
1463      buffer = OPENSSL_malloc(1 + 2 + payload + padding);
1464      bp = buffer;
1465      /* Enter response type, length and copy payload */
1466      *bp++ = TLS1_HB_RESPONSE;
1467      *bp++ = payload;
1468      memcpy(bp, pl, payload);
1469    }
```

```
/* Naive implementation of memcpy */
void *memcpy(void *dst, const void *src, size_t n)
{
  size_t i;
  for (i=0; i<n; i++)
    *(char *) dst++ = *(char *) src++;
  return dst;
}
```
Clear Causality in Heartbleed

DVR
(Verify: Missing, Data (payload length), Policy)

MAD
(Reposition, Pointer, Object, Size: Inconsistent Value, Size)

MUS
(Clear: Missing, Pointer, Object, Size)

Information Exposure

Inconsistent Value

Over Bounds

Buffer Overflow

Buffer Overflow

Not Cleared Object

Caused by the Bug

Caused by ill-formed data

The Failure – caused by final error(s)
The Bug
A Weakness
The Failure
BF Tool – Generated Machine-Readable BF Heartbleed Description

CVE-2014-016...Overflow.bf

```xml
<Vulnerability Name="Buffer Overflow">
  <Bug Type="_INP" Class="DVR">
    <Cause Type="Improper Operation" Comment=""">Missing</Cause>
    <Operation Comment=""">Verify</Operation>
    <Consequence Type="Improper Data Value" Comment="for payload size">Inconsistent Value</Consequence>
    <Attributes>
      <Operation>
        <Attribute Type="Mechanism">Quantity</Attribute>
        <Attribute Type="Source Code">Codebase</Attribute>
        <Attribute Type="Execution Space" Comment="">Transfer</Attribute>
      </Operation>
  </Attributes>
  <Weakness Type="_MEM" Class="MUS">
    <Cause Type="Improper Address" Comment="for s+=s3+rrec.data[0]">Over Bounds Pointer</Cause>
    <Operation Comment="">Read</Operation>
    <Consequence Type="Memory Error" Comment="">Buffer Overflow</Consequence>
    <Attributes>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Type="Source Code">Codebase</Attribute>
        <Attribute Type="Execution Space">UserLand</Attribute>
      </Operation>
      <Operand Name="Address">
        <Attribute Type="Span">Huge</Attribute>
        <Attribute Type="Location">Heap</Attribute>
      </Operand>
    </Attributes>
  </Weakness>
  <Weakness Type="_MEM" Class="MAD">
    <Cause Type="Improper Data Value" Comment="for size">Over Reposition</Cause>
    <Operation Comment="">Reposition</Operation>
    <Consequence Type="Improper Address" Comment="">Over Reposition</Consequence>
    <Attributes>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Type="Source Code">Codebase</Attribute>
        <Attribute Type="Execution Space">UserLand</Attribute>
      </Operation>
      <Operand Name="Address">
        <Attribute Type="Span">Huge</Attribute>
        <Attribute Type="Location">Heap</Attribute>
      </Operand>
    </Attributes>
  </Weakness>
</Bug>
```

Failure Type="_XXX" Class="IEX">
  <Cause Type="Memory Error" Comment="">Buffer Overflow</Cause>
  <Operation Comment="">IEX Operation</Operation>
  <Consequence Type="Risk" Comment="">IEX Consequence</Consequence>
</Failure>
</Vulnerability>
```
Towards a “Periodic Table” of Bugs

Irena Bojanova, Paul E. Black, Yaacov Yesha, Yan Wu

April 9, 2015

NIST, BGSU

- Heartbleed buffer overflow is:
  - caused by Data Too Big
  - because of User Input not Checked Properly
  - where there was a Read that was After the End that was Far Outside
  - of a buffer in the Heap
  - which may be exploited for Information Exposure

Input not checked properly leads to too much data, where a huge number of bytes are read from the heap in a continuous reach after the array end, which may be exploited for exposure of information that had not been cleared.
The Bugs Framework (BF)
Early Work

They Know Your Weaknesses
Reintroducing Common Weaknesses

Irena Bojanova
UMUC, NIST

12/08/2014

CWE-128 in Z notation

CWE-128: Wrap-around Error: “Wrap around errors occur in incremented past the maximum value for its type and the error is caused to a very small, negative, or undefined value.”

\[
\text{MAX_INT} = 2^{31} - 1 = 2147483647
\]

\[
\text{MIN_INT} = -2^{31} = -2147483648
\]

\[
\text{BAD_INT} = \text{MIN_INT} + \text{MAX_INT} < \text{BAD_INT}
\]

\[
\text{BAD_INT < MIN_INT} \text{ or } \text{MAX_INT < BAD_INT}
\]

CWE-2014-160/CAPEC-540 in CSP

channel network 2:

\[
\text{enum } \text{payloadLength, payload, validPayload, invalidPayload);}
\]
\[
\text{Attacker()} = \text{network} \text{payloadLength} \text{-> network} \text{payload} \text{->}
\]
\[
\text{network} \text{payloadResponse} \text{-> attacker}();
\]
\[
\text{CWE126}() = \text{network} \text{payloadLength} \text{-> network} \text{payload} \text{->}
\]
\[
\text{(payloadLength} \text{!= payloadLengthSize} \text{-> network} \text{validPayload} \text{-> CWE126()}}
\]
\[
\text{(() payloadLength} \text{!= payloadLengthSize} \text{-> network} \text{invalidPayload} \text{->}
\]
\[
\text{CWE126();}
\]

System() = Attacker() || CWE126();

Towards a “Periodic Table” of Bugs

Irena Bojanova, Paul E. Black, Yaakov Yeshua, Yan Wu

April 9 – July 23, 2015

NIST, BGSU
The Bugs Framework (BF): A Structured Approach to Express Bugs

Paul E. Black
SIT
Gatlinburg, USA
black@siteseer.net

Yaya Yvon
NST
Gatlinburg, USA, Baltimore, USA
yyvon@siteseer.net

Yen Wu
Bingding Gow
Boston, USA
ywu@siteseer.net

The Bugs Framework (BF) provides a structured approach for expressing bugs in software. It is based on a framework that identifies and describes software weaknesses and vulnerabilities. The BF allows developers to express bugs in a structured manner, making it easier to understand and analyze them.

1. Introduction

The BF is designed to provide a structured approach for expressing bugs in software. It identifies and describes software weaknesses and vulnerabilities. The BF allows developers to express bugs in a structured manner, making it easier to understand and analyze them.

2. Description

The BF is designed to provide a structured approach for expressing bugs in software. It identifies and describes software weaknesses and vulnerabilities. The BF allows developers to express bugs in a structured manner, making it easier to understand and analyze them.

3. Applications

The BF is designed to provide a structured approach for expressing bugs in software. It identifies and describes software weaknesses and vulnerabilities. The BF allows developers to express bugs in a structured manner, making it easier to understand and analyze them.

4. Conclusion

The BF is designed to provide a structured approach for expressing bugs in software. It identifies and describes software weaknesses and vulnerabilities. The BF allows developers to express bugs in a structured manner, making it easier to understand and analyze them.
Missing Cornerstones

● Strict Definitions of:
  ○ Bug
  ○ Weakness
  ○ Vulnerability
  ○ Failure

● Clarity on:
  ○ Chaining Bugs/Weaknesses/Failures
  ○ Merging Chains
Terminology

- **Software Bug:**
  - A coding error
  - Needs to be fixed

- **Software Weakness – difficult to define:**
  - Caused by a bug or ill-formed data
  - Weakness Type – a meaningful notion!

- **Software Vulnerability:**
  - An instance of a weakness type that leads to a security failure
  - May have several underlying weaknesses

- **Security failure:**
  - A violation of a system security requirement

BF Goals

1. Solve the problems of imprecise descriptions and unclear causality

2. Solve the problems of gaps and overlaps in coverage
BF describes a bug/weakness as:

- An improper state
- Its transition

**Improper State** – a tuple \((\text{operation}, \text{operand}_1, \ldots, \text{operand}_n)\), where at least one element is improper

**Transition** – the result of the operation over the operands

BF Features – Clear Causal Descriptions

- Initial State – caused by the Bug
- Intermediate State – caused by ill-formed data
- Final State – the Failure

Failure

Initial State – caused by the Bug

Intermediate State – caused by ill-formed data

Final State – the Failure

- caused by a final error
BF Features – Chaining Weaknesses

- BF describes a vulnerability as:
  - A chain of improper states and their transitions
  - States change until a failure is reached
BF Features – Backtracking

- How to find the Bug?
- Go backwards by operand until an operation is a cause
BF Features – Converging Vulnerabilities

- **Improper State 1**: (operation 1, operand 1, ... operand 1, ...)
- **Improper State 1’**: (operation 1’, operand 1’, ... operand 1’, ...)
- **Improper State 2**:
- **Improper State n**: (operation n, operand n, ... operand n, ...)
- **Improper State q’**: (operation q’, ... operand q’, ...)
- **Final Error**

**Initial State**: caused by the Bug – the operation is improper
**Intermediate State**: caused by ill-formed data – at least one operand is improper
**Final State**: the Failure – caused by a final error
BF Features – Classification

- BF Class – a taxonomic category of a weakness type, defined by:
  - A set of operations
  - All valid cause → consequence relations
  - A set of attributes
**BF Features – Tools**

- **Creation of:**
  - BF classes diagrams
  - BF-CWE di-graphs
  - Vulnerabilities graphs & diagrams

- **Querying of:**
  - Vulnerabilities

---

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Operation</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanism:</td>
<td>Source Code:</td>
<td>Execution Space:</td>
</tr>
<tr>
<td>Range</td>
<td>Codebase</td>
<td>Userland</td>
</tr>
<tr>
<td>Is Null</td>
<td>Third Party</td>
<td>Kernel</td>
</tr>
<tr>
<td>Safe List</td>
<td>Standard Library</td>
<td>Bare-Metal</td>
</tr>
<tr>
<td>Unsafe List</td>
<td>Processor</td>
<td>In Use</td>
</tr>
<tr>
<td>Business Logic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Cause**

- Improper Operation: Missing

**DRI Operation**

- Validate

**Consequence**

- Injection Error: File Injection (Relative Path Traversal)

**Attributes**

- Operation
  - Source Code: Codebase (login.jsp)
  - Execution Space: Admin
- Data
  - State: Transferred (via network)
BF – Defined

- BF is a ...

  - Structured
  - Complete
  - Orthogonal
  - Language independent

classification of software bugs and weaknesses
BF – Bugs Models

- Example:

The BF Memory Bugs Model:

- Four phases, corresponding to the BF memory bugs classes: MAD, MAL, MUS, MDL
- Memory operations flow
BF Classes – Examples

Data Verification Bugs (DVR)
- Causes:
  - Improper Operation: Missing, Erroneous
  - Improper Data: Corrupted, Unauthentic, Wrong Number, Wrong Data Type, Meaningless
- DVR Operations:
  - Verify
  - Sanitize Semantics
- Consequences:
  - Improper Operation: Missing, Mismatched, Erroneous

Memory Addressing Bugs (MAD)
- Causes:
  - Improper Operation: Missing, Mismatched, Erroneous
- MAD Operations:
  - Initialize
  - Reposition
  - Reassign
- Consequences:

Memory Use Bugs (MUS)
- Causes:
  - Improper Operation: Missing, Mismatched, Erroneous
- MUS Operations:
  - Initialize
  - Dereference
  - Read
  - Write
  - Clear
- Consequences:

Memory Error:
- Uninitialized Object
- Not Cleared Object
- NULL Pointer Dereference
- Untrusted Pointer Dereference
- Object Corruption
- Type Confusion
- Use After Free
- Buffer Overflow
- Buffer Underflow
- Uninitialized Pointer Dereference

Object Attributes:
- Improper Object: Wrong Size Used, Not Enough Allocated
- Attributes:
  - Direct, Sequential
  - Operation: Stack, Heap, ...
BF – Validation Towards CWE

- Input/Output CWEs (incl. Injection) – mapped by BF DVL and BF DVR consequences

CWE by DVL Injection Error:
- Query Injection
- Command Injection
- Source Code Injection
- Parameter Injection
- File Injection

CWE by Abstraction:
- Pillar
- Class
- Base
- Variant
- Compound

CWE by DVL or DVR Wrong Data for Next Operation Consequence:
- DVL Invalid Data
- DVR Wrong Value, Inconsistent Value, and Wrong Type
- No consequence (only cause listed)
Example:

BF Chain for “BadAlloc” Pattern
ICS Advisory (ICSA-21-119-04)

Multiple RTOS (Update E)

Original release date: April 19, 2022

Legal Notice
All Information products included in https://us-cert.cisa.gov/ics are provided "as is" for informational purposes only. The Department of</p>

1. EXECUTIVE SUMMARY
- CVSS v3 9.8
- ATTENTION: Exploitable remotely/low attack complexity
- Vendors: Multiple

4.2 VULNERABILITY OVERVIEW
4.2.1 INTEGER OVERFLOW OR WRAPAROUND CWE-190
MediaTek Linkit SDK versions prior to 4.6.1 is vulnerable to integer overflow in memory allocation calls pPortCalloctab() and memory corruption on the target device.
CVE-2021-30638 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vector is (AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:H).

4.2.2 INTEGER OVERFLOW OR WRAPAROUND CWE-190
ARM CMSISRTOS2 versions prior to 2.1.3 are vulnerable to integer wrap-around in osRxMemoryAlloc (local malloc equivalent), resulting in unexpected behavior such as a crash or injected code execution.
CVE-2021-27431 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vector is (AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:H).

4.2.3 INTEGER OVERFLOW OR WRAPAROUND CWE-190
ARM mbed-alloc memory library Version 1.3.0 is vulnerable to integer wrap-around in function mbed_ktib, which can lead to unexpected behavior such as a crash or remote code injection/execution.
CVE-2021-27433 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vector is (AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:H).

4.2.4 INTEGER OVERFLOW OR WRAPAROUND CWE-190
ARM mbed product Version 6.3.0 is vulnerable to integer wrap-around in malloc_wrapper function, which can lead to unexpected behavior such as a crash or remote code injection/execution.
CVE-2021-27435 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vector is (AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:H).

4.2.5 INTEGER OVERFLOW OR WRAPAROUND CWE-190
RIOT OS Versions 2020.01.1 is vulnerable to integer wrap-around in its implementation of calloc function, which can lead to unexpected behavior such as a crash or remote code injection/execution.
CVE-2021-27427 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vector is (AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:H).
CVE-2021-21834 and the Bad Allocation Chain

**DVR Operation**
- **Cause**: Improper Operation: Missing
- **Mechanism**: Range
- **Source Code**: Third Party (library box_code_base.c)
- **Execution Space**: Local
- **Data State**: Stored (number of entries read from file)
- **Consequence**: Improper Data Value: Inconsistent Value (> max 64-bit int)

**TCM Operation**
- **Cause**: Improper Data Value: Wrong Argument
- **Mechanism**: Operator (Arithmetic: ‘*’)
- **Source Code**: Third Party (library box_code_base.c)
- **Data Value Kind**: Numeric
- **Data Type Kind**: Structured
- **Execution Space**: Userland
- **Object Location**: Heap
- **Pointer Ownership**: Single
- **Consequence**: Improper Object Size: Not Enough Allocated

**MAL Operation**
- **Cause**: Improper Data Value: Wrong Size
- **Mechanism**: Explicit
- **Source Code**: Third Party (library box_code_base.c)
- **Execution Space**: Userland
- **Object Location**: Heap
- **Consequence**: Improper Object Size: Not Enough Allocated

**MAD Operation**
- **Cause**: Improper Object Size: Not Enough Allocated
- **Mechanism**: Sequential
- **Source Code**: Third Party (library box_code_base.c)
- **Execution Space**: Userland
- **Object Location**: Heap
- **Pointer Span**: Huge
- **Consequence**: Improper Data Value: Over Bounds Pointer

**MUS Operation**
- **Cause**: Improper Data Value: Over Bounds Pointer
- **Mechanism**: Sequential
- **Source Code**: Third Party (library box_code_base.c)
- **Execution Space**: Userland
- **Object Location**: Heap
- **Pointer Span**: Huge
- **Consequence**: Memory Error: Buffer Overflow
BF Tools Set
The BF vulnerabilities descriptions consist of bug-weaknesses-failure chains. This tool would allow users to:

1. To create instances of bugs, weaknesses, and failures with specific cause, operation, and consequence selections, connect these instances by consequence-cause relationships, and specify attributes about each involved operation and its operands. The resulting BF vulnerabilities' descriptions will be in an XML .bf format adhering to a BF Vulnerability description XSD schema.

2. To generate graphical PPTX representations of BF vulnerabilities descriptions via XSLT transformations.

3. To edit and query generated BF vulnerabilities descriptions.
1. BF.xml – all BF Clusters of Classes
1. BF Editor

Bug/Weakness/Failure

Vulnerability:
- DVR
- MAD
- MUS
- IEX

BF Class:
- JNP
- DVL
- DVR
- DTC
- DTT
- TUS
- TCC
- MEM
- MAD
- MAL
- MUS
- MDL
- CR
- ENC
- VR
- VRF
- MRN
- RND
- TRN
- PRN

Weakness:
- Over Bounds Pointer

Preceding Consequence:
- Over Bounds Pointer

Consequence:
- Memory Error
- Uninitialized Object
- Not Cleared Object
- NULL Pointer Dereference
- Untrusted Pointer Dereference
- Object Corruption
- Type Confusion
- Use After Free
- Buffer Overflow
- Buffer Underflow
- Uninitialized Pointer Dereference

Cause:
- Improper Data Value
  - Forbidden Address
  - Wrong Size Used
- Improper Data Type
  - Casted Pointer
  - NULL Pointer
  - Wild Pointer
  - Dangling Pointer
  - Untrusted Pointer
  - Under Bounds Pointer
  - Wrong Position Pointer

Operation:
- Initialize
- Dereference
- Read
- Write
- Clear

Operand Attributes:
- Buffer Overflow

Operand Attributes:
- Address
  - Relative
  - Relative
- Scale
  - Scale
  - Scale
- Location
  - Stack
  - Heap
  - Other

Comment:
- for s="s3"-rec.data[0]

Following Cause:

Comment:

Operand Attributes:
- Buffer Overflow

Operation Attributes:
- Mechanism
  - Direct
- Sequential
- Source Code
  - Codebase
- Third Party
  - Standard Library
  - Language Processor
- Execution Space
  - Userland
  - Kernel
  - Bare-Metal

Comment:
- d__both.c and tli_lib.c
2. Generated Graphical Representation of BF Heartbleed Description

DVR (Verify: Missing, Data (payload length), Policy)

Inconsistent Value

MAD (Reposition, Pointer, Object, Size: Inconsistent Value)

Over Bounds Pointer

MUS (Read, Pointer: Over Bounds, Object, Size)

Buffer Overflow

IEX (Buffer Overflow)

Not Cleared Object

Caused by the Bug

Caused by ill-formed data

The Failure – caused by final error(s)
2. Detailed Graphical Representation of the BF Heartbleed Description

**Cause**
- Improper Operation: Missing
- Inconsistent Value: Wrong Size Used (for s.s3.rrec.data[0])
- Improper Address: Over Bounds Pointer

**DVR Operation**
- Verify

**Consequence**
- Improper Data Value: Inconsistent Value (for payload size)
- Improper Address: Over Bounds Pointer

**MAD Operation**
- Reposition

**MUS Operation**
- Read

**Consequence**
- Improper Data Value: Not Cleared Object
- Memory Error: Buffer Overflow

**Attributes**
- Operation
  - Mechanism: Sequential
  - Source Code: Codebase (d1_both.c and tl_lib.c)
  - Execution Space: Userland
  - Location: Heap

- Data
  - State: Transferred (via network)

- Operation
  - Mechanism: Sequential
  - Source Code: Codebase (other software)
  - Execution Space: Userland
  - Span: Huge
  - Location: Heap

- Operation
  - Mechanism: Sequential
  - Source Code: Codebase (d1_both.c and tl_lib.c and other software)
  - Execution Space: Userland
  - Span: Huge
  - Location: Heap

---

The Bug
A Weakness
The Failure
3. Edit and Query BF Descriptions

- Edit generated BF vulnerabilities descriptions.
- Allow BF vulnerabilities descriptions that converge two or more chains via "and/or" conjunctions.
- Query BF vulnerabilities' descriptions by:
  - ✓ Class
  - ✓ Operation
  - ✓ Cause
  - ✓ Consequence
  - ✓ Attributes
  - ✓ and combinations of such.
II. Editor of BF Classes and BF Clusters

This tool will allow BF developers collaborators to:

1. To create descriptions of BF classes with sets of values for each class causes, operations, and consequences, as well as of matrices with meaningful cause-operation-consequences combinations. The resulting BF classes descriptions will be in XML format adhering to a BF classes XSD schema and can be used by software assurance tools developers to report found bugs and weaknesses, as well as to provide precise vulnerabilities' descriptions.

2. To generate graphical representations of BF classes from the XML descriptions via XSLT transformations. The graphical representations will be in PowerPoint .pptx format.

3. To generate BF-CWEs relational di-graphs for validation of newly developed BF classes towards the flawed, but widely used CWE.
1. BF Classes and Matrices of Cause-Operation-Consequences

- Create descriptions of BF classes with sets of values for each class causes, operations, and consequences.

- Create matrices with meaningful cause-operation-consequences combinations.
2. Generated Graphical Representations of the BF TCV & TCM Classes

Type Conversion Bugs (TCV)

- **Causes**
  - Improper Operation:
    - Missing
    - Wrong
  - Improper Data Value:
    - Under Range
    - Over Range
    - Flipped Sign
  - Improper Data Type:
    - Wrong Type
  - Improper Function Implementation:
    - Missing Overload

- **TCV Operations**
  - Cast
  - Coerce

- **Consequences**
  - Improper Data Value:
    - Wrong Result
    - Truncated Value
    - Distorted Value
    - Rounded Value
  - Improper Data Type:
    - Wrong Type

- **Attributes**
  - Mechanism: Pass In, Pass Out
  - Source Code: Codebase, Third Party, Standard Library, Processor
  - Data Value Kind: Numeric, Text, Pointer, Boolean
  - Data Type Kind: Primitive, Structured

Type Compute Bugs (TCM)

- **Causes**
  - Improper Operation:
    - Wrong
    - Erroneous
  - Improper Data Value:
    - Under Range
    - Over Range
    - Flipped Sign
    - Wrong Result
    - Wrap Around
  - Improper Data Type:
    - Wrong Type
  - Improper Function Implementation:
    - Wrong Function
    - Wrong Generic Function
    - Wrong Overridden Function
    - Wrong Overloaded Function

- **TCM Operations**
  - Calculate
  - Evaluate

- **Consequences**
  - Improper Data Value:
    - Wrong Object
    - Reference vs. Object
    - Wrong Argument
  - Improper Data Type:
    - Wrong Type
  - Improper Function Implementation:
    - Wrong Function
    - Wrong Generic Function
    - Wrong Overridden Function
    - Wrong Overloaded Function

- **Type Compute Error:** Undefined (div by 0)

- **Attributes**
  - Mechanism: Function, Operator, Method, Lambda Expression, Procedure
  - Source Code: Codebase, Third Party, Standard Library, Processor
  - Data Value Kind: Numeric, Text, Pointer, Boolean
  - Data Type Kind: Primitive, Structured
3. CWEs Relate to BF Clusters
Generated Graphical Representations of the Input/Output Cluster Mappings to CWE
BF – Potential Impact
BF – Potential Impacts

- Allow precise communication about software bugs and weaknesses
- Help identify exploit mitigation techniques
BF Addresses a Unique Need

- JHU APL – **Automated Vulnerability Testing via Executable Attack Graphs**:
  - Chain vulnerabilities via logical directed graphs
  - Determine most mitigation “paths” with least changes
  - Detect user behavior prior to malicious effect

The lack of formal, precise descriptions of known vulnerabilities and software weaknesses in the current National Vulnerability Database (NVD) has become an increasingly limiting factor in vulnerability research, mitigation research, and expression of software systems in low level modeling form.

A critical need for this research is a reliable set of well-formalized expressions that are machine-ingestible. Dr. Bojanova’s proposed **BF Tool Set** would allow the creation of well-formed descriptors for the software weaknesses, the vulnerabilities that can be exploited, and the failures/effects that can be realized for each bug. Such a repository of information could be ingested by all researchers looking to explore complex chains of vulnerabilities, which comprise the vast majority of malicious cyber incidents worldwide.

We were thrilled to hear that a researcher at NIST was undertaking the needed improvement to make such descriptions more formal and machine-readable. Such an endeavor will greatly enhance the ability of cyber researchers to explore more complex attacks via computational methods. This will provide a huge boost to the U.S.’s ability to defend its networks, military systems, and critical infrastructure, and will lead the way to better mitigation designs, improved software development practices, and automated cyber testing capabilities.
BF Addresses a Unique Need

- RIT Secure and Trustworthy Cyberspace (SaTC):
  - Projects on Vulnerabilities Research

The NIST Bugs Framework (BF) has made significant advances in creating first-of-its-kind classification of software weaknesses that has enabled the community to express vulnerabilities using a precise description, allowing us to obtain a fine-grained understanding of security bugs and their root causes. Additionally, the taxonomies and root causes in each bug class will provide us valuable data to guide and enhance our static program analysis techniques and achieve higher accuracy.

Supports various research initiations at DARPA and other agencies. For instance, the notion of “Weird Machines”- unintended, emergent program behaviors and attack scenarios in DARPA’s Artificial Intelligence Mitigations of Emergent Execution (AIMEE) program can be better explained and tamed using BF classes that capture such complex root causes.

Bugs Framework (BF) Tools Set can bring the software security community together in better understanding of software security bugs but also development of high-fidelity tools.
More Interest and Support

- INMETRO
- LLNL
- BIECO
- Fraunhofer IESE
- CSA
- University of Greenwich
- Carnegie Mellon University
- St. John’s University
- University of West Attica
- Ericsson
- Anchore Inc.
Classifying Memory Bugs Using Bugs Framework Approach

Jenni Bogdanova
SSD, IIT
Glasgow, MD, USA
jenni.bogdanova@iit.edu

Carlos Eduardo Galhardo
Dadad, Brasil
Carlos@dada.com, B, Brasil
carlod@gmail.com

In this paper we present a novel approach for identifying and categorizing memory bugs, allowing precise software developers and system engineers to efficiently remove memory bugs from their code. We propose a new algorithm called Memory Bugs Framework Approach (MBFA) that uses a combination of static and dynamic analysis techniques to identify memory bugs. MBFA first identifies potential memory bugs by analyzing memory allocation and deallocation patterns. It then uses a combination of static analysis techniques to identify the precise location of the memory bug, as well as the conditions under which the bug is likely to occur. Finally, MBFA uses dynamic analysis techniques to confirm the presence of the memory bug and to determine the severity of the bug. MBFA provides a comprehensive and efficient solution for identifying and eliminating memory bugs in software systems.

II. INTRODUCTION

Software bugs in memory allocation, use, and deinitialization can lead to memory corruption and memory exhaustion, opening doors for cyberattacks. Classifying these bugs would allow for better network communication and help to reach about their under-stand and identify them, and avoid security failures. Therefore, we utilize the Bug Framework Approach (MBFA).

The Common Weakness Enumeration (CWE) [1] and the Common Vulnerabilities and Exposures (CVE) [2] are well known and used lists of software security weaknesses and vulnerabilities. However, the MBFA classification approach is focused on bugs being prone to leaks and usage in misuse, as demonstrated by the National Vulnerability Database (NVD). MBFA is not to be conflated with CVEs or with reported software vulnerabilities.

In this paper, we utilize the MBFA approach to analyze software types, abstract language-independent, software classes that cover all possible kinds of memory related bugs and vulnerabilities. Memory Allocation Bugs (MAL), Memory Use Bugs (MUB), Memory Deinitialization Bugs (MDI), Memory Allocating Addressing Bugs (MAA), and Memory Bugs Taxonomy can be viewed as a structured enumeration and categorization of memory-related bugs, allowing bug reporting tools to produce more detailed, precise, and meaningful information about memory bugs than the traditional CWE/CVE classification approach.

Declaration: Certain tools and computer products mentioned in the text or used in the paper are not considered to be a endorsement by the National Institute of Standards and Technology (NIST). They are merely cited for the purpose of providing further insight into the research presented in the paper.

III. RELATED WORK

In this section, we compare the MBFA approach with other memory bug classification approaches. We discuss the advantages and disadvantages of each approach and how MBFA improves upon them.

IV. METHODOLOGY

In the methodology section, we describe the process used to identify and categorize memory bugs using the MBFA approach. We provide a detailed overview of the steps involved in the methodology, including data collection, analysis, and validation.

V. RESULTS

In the results section, we present the findings of our analysis of memory bugs using the MBFA approach. We discuss the prevalence of different types of memory bugs and their impact on software systems.

VI. DISCUSSION

In this section, we discuss the implications of our findings and the potential for future research in memory bug classification.

VII. CONCLUSION

In conclusion, we present the MBFA approach as a novel and efficient method for identifying and classifying memory bugs. Our approach provides a comprehensive and detailed analysis of memory bugs, allowing software developers and system engineers to efficiently eliminate memory bugs from their code. We believe that the MBFA approach has significant potential for improving the security of software systems by providing a more detailed and precise classification of memory bugs.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES


Questions

Irena Bojanova: irena.bojanova@nist.gov