The NIST
Bugs Framework (BF)

https://samate.nist.gov/BF/

National Institute of
Standards and Technology
U.S. Department of Commerce

Irena Bojanova

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

Problem #1: Imprecise Descriptions

- Example:

  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

Problems #2, #3: Unclear Causality, Tracking

- 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

	Over	Under	Either End		Stack	Heap
Read	CWE-127	CWE-126	CWE-125		✦	✦
Write	CWE-124	CWE-120	CWE-123 CWE-787 ✦		CWE-121	CWE-122
Read/ Write	CWE-786	CWE-788	✦		✦	✦

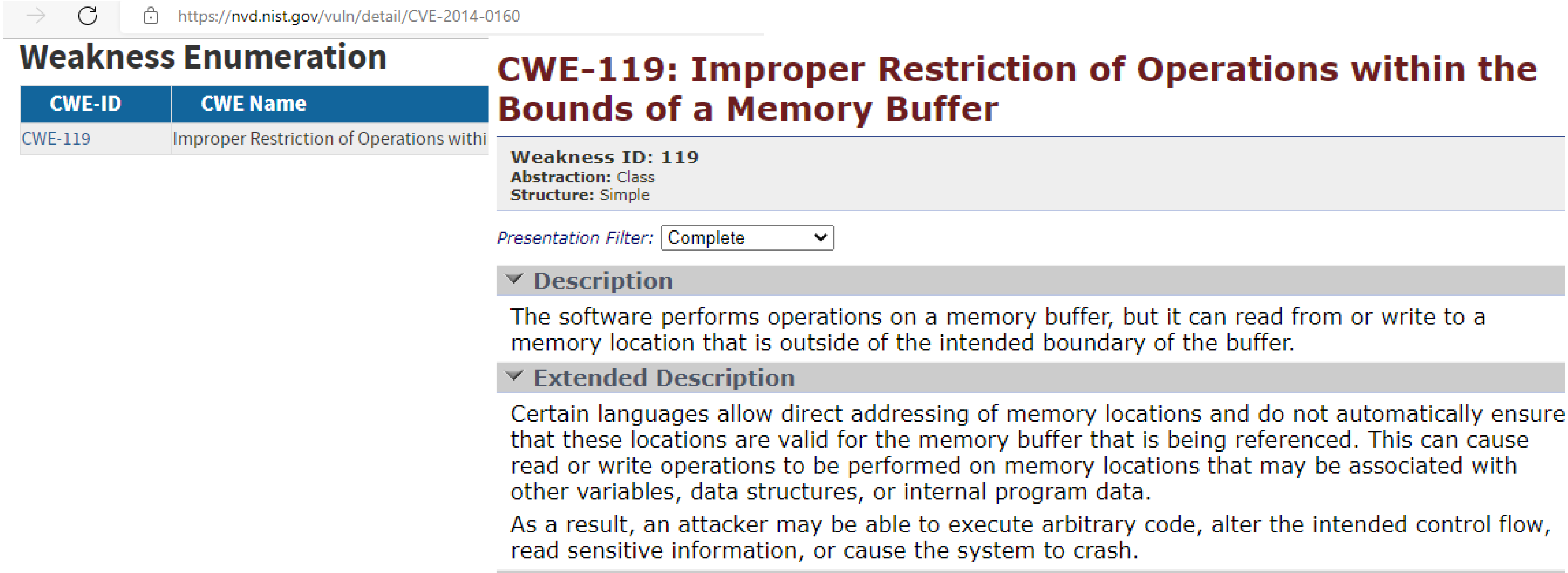
The Bugs Framework (BF)

Example:

CVE versus BF Descriptions of Heartbleed

Heartbleed (CVE-2014-0160)

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.



https://nvd.nist.gov/vuln/detail/CVE-2014-0160

Weakness Enumeration

CWE-ID	CWE Name
CWE-119	Improper Restriction of Operations withi

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

Weakness ID: 119
Abstraction: Class
Structure: Simple

Presentation Filter: Complete

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.

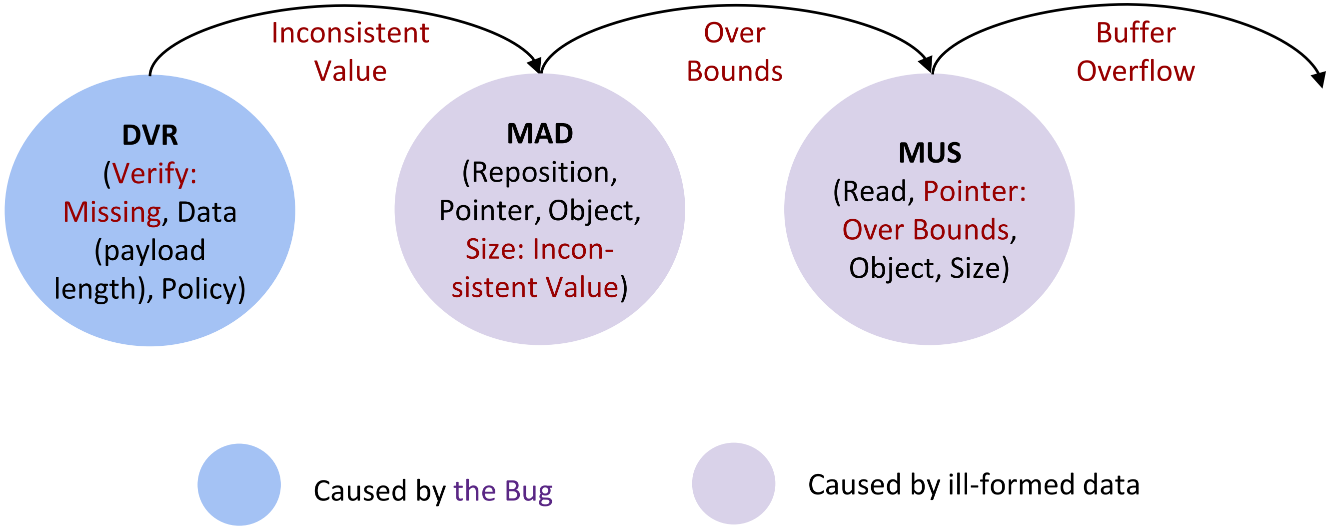
As a result, an attacker may be able to execute arbitrary code, alter the intended control flow, read sensitive information, or cause the system to crash.

Heartbleed (CVE-2014-0160)

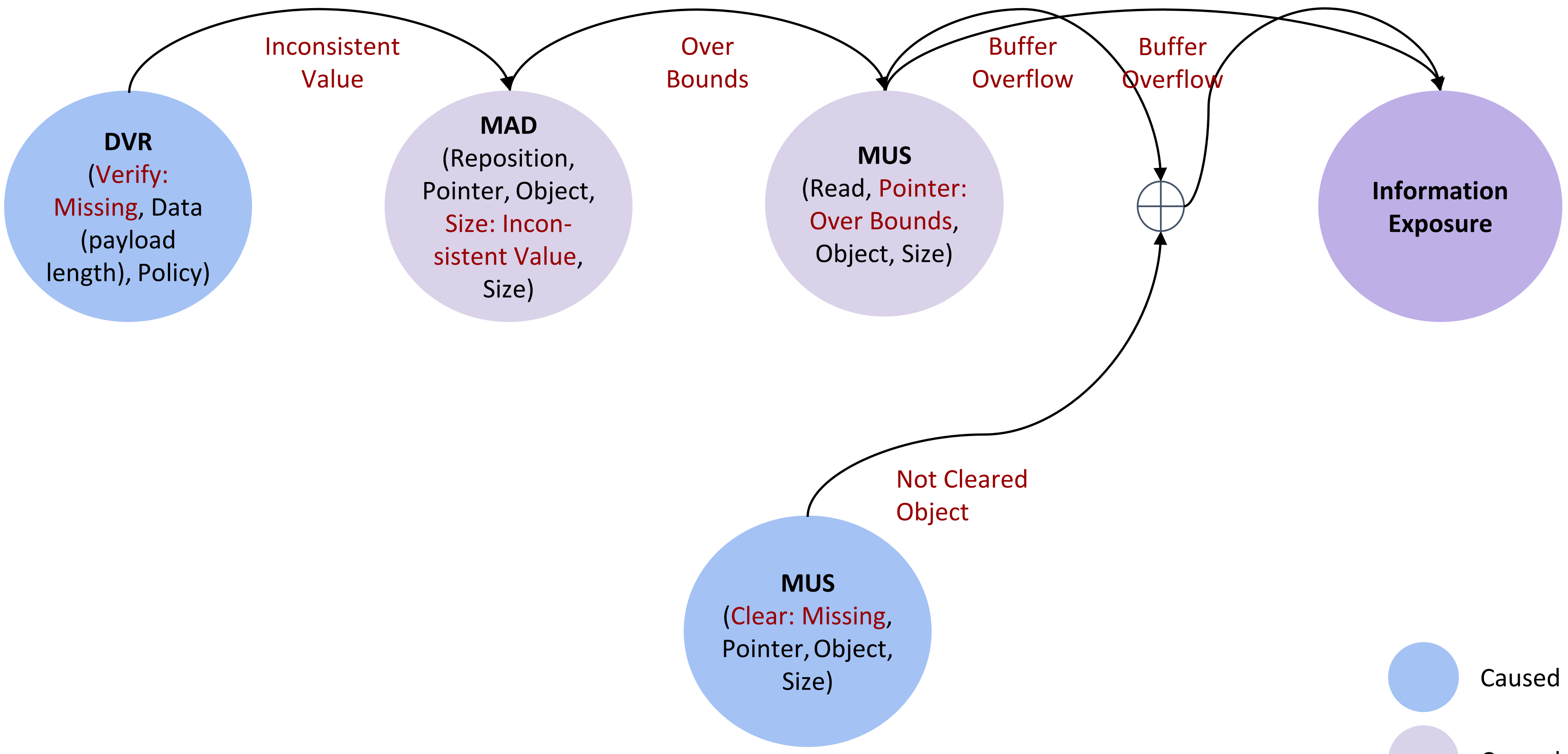
CVE-2014-0160 The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private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

```
dtls1_process_heartbeat(SSL *s)
  {
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;
...
  if (hbtype == TLS1_HB_REQUEST)
     {
     unsigned char *buffer, *bp;
...
     /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload, plus
     * payload, plus padding
     */
     buffer = OPENSSL_malloc(1 + 2 + payload + padding);
     bp = buffer;

     /* Enter response type, length and copy payload */
     *bp++ = TLS1_HB_RESPONSE;
     s2n(payload, bp);
     memcpy(bp, pl, payload);
```

```
/* Naive implementation of memcpy
void *memcpy (void *dst, const void *src, size_t n)
{
    size_t i;        payload
    for (i=0; i<n; i++)
        *(char *) dst++ = *(char *) src++;
    return dst;
}                  bp                  pl
```



Clear Causality in Heartbleed



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

Inconsistent Value

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

Over Bounds

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

Buffer Overflow

Buffer Overflow

Information Exposure

Not Cleared Object

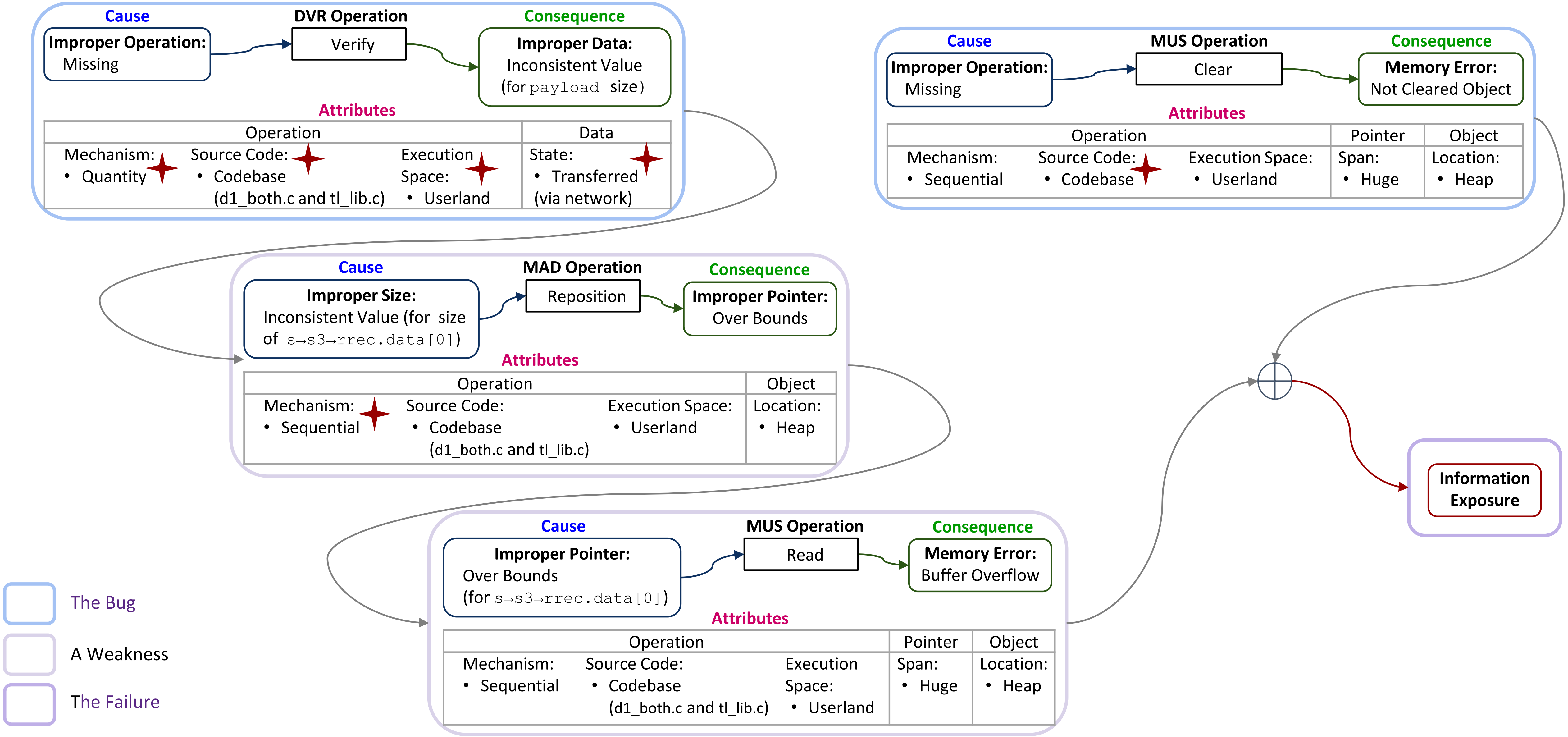
MUS
(Clear: Missing, Pointer, Object, Size)

● Caused by the Bug

● Caused by ill-formed data

● The Failure – caused by final error(s)

BF Description of Heartbleed



Cause

Improper Operation: Missing

DVR Operation

Verify

Consequence

Improper Data: Inconsistent Value (for `payload` size)

Attributes

Operation			Data
Mechanism: • Quantity	Source Code: • Codebase (d1_both.c and tl_lib.c)	Execution Space: • Userland	State: • Transferred (via network)

Cause

Improper Operation: Missing

MUS Operation

Clear

Consequence

Memory Error: Not Cleared Object

Attributes

Operation			Pointer	Object
Mechanism: • Sequential	Source Code: • Codebase	Execution Space: • Userland	Span: • Huge	Location: • Heap

Cause

Improper Size: Inconsistent Value (for size of `s→s3→rrec.data[0]`)

MAD Operation

Reposition

Consequence

Improper Pointer: Over Bounds

Attributes

Operation			Object
Mechanism: • Sequential	Source Code: • Codebase (d1_both.c and tl_lib.c)	Execution Space: • Userland	Location: • Heap

Cause

Improper Pointer: Over Bounds (for `s→s3→rrec.data[0]`)

MUS Operation

Read

Consequence

Memory Error: Buffer Overflow

Attributes

Operation			Pointer	Object
Mechanism: • Sequential	Source Code: • Codebase (d1_both.c and tl_lib.c)	Execution Space: • Userland	Span: • Huge	Location: • Heap

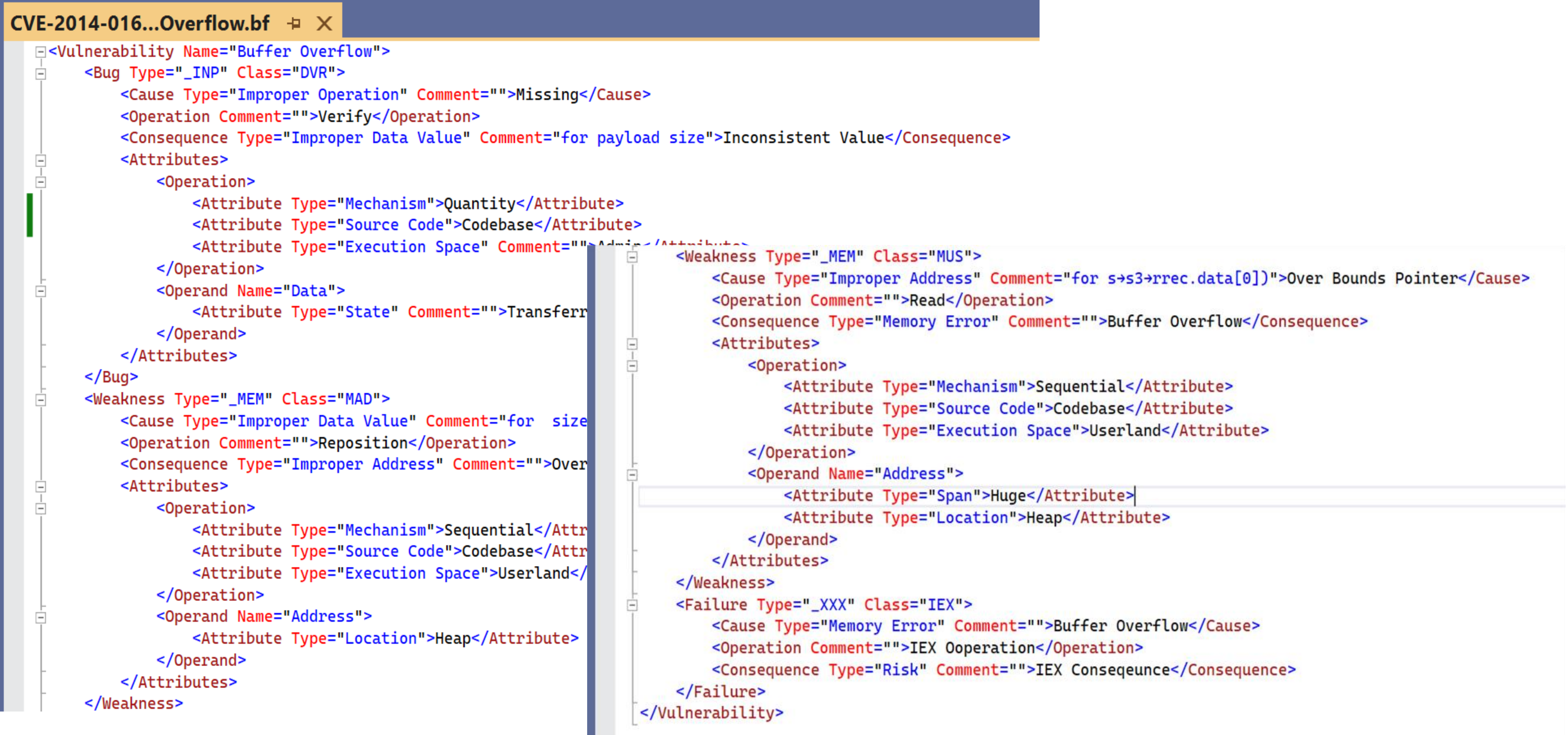
Information Exposure

The Bug

A Weakness

The Failure

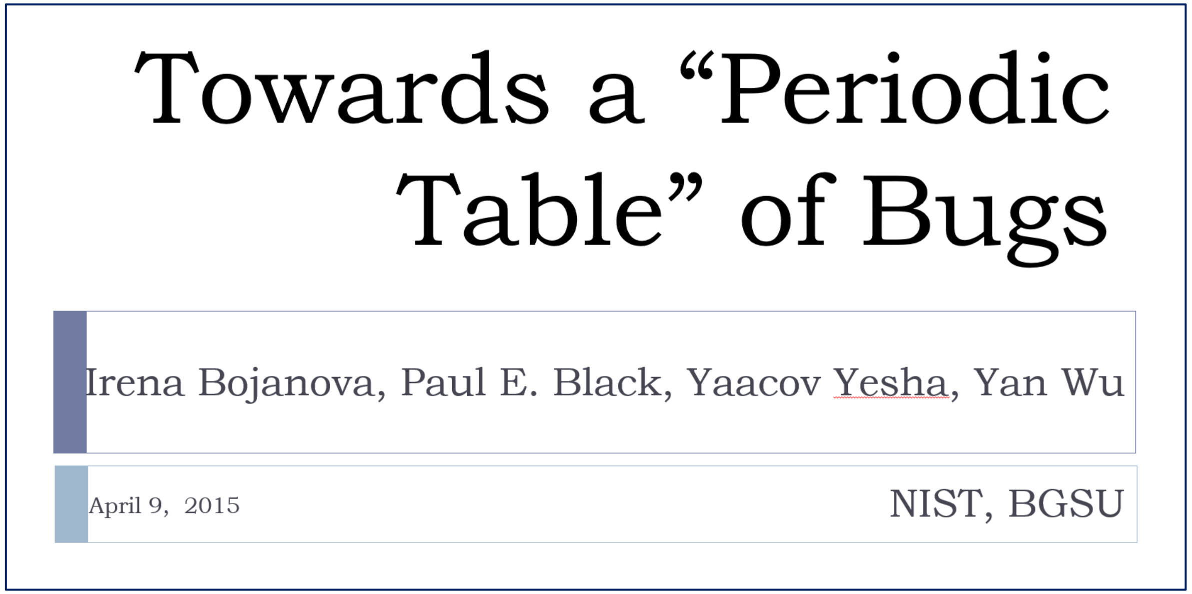
BF Tool – Generated Machine-Readable BF Heartbleed Description



```
<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="">Admin</Attribute>
            </Operation>
            <Operand Name="Data">
                <Attribute Type="State" Comment="">Transferr
            </Operand>
        </Attributes>
    </Bug>
    <Weakness Type="_MEM" Class="MAD">
        <Cause Type="Improper Data Value" Comment="for  size
        <Operation Comment="">Reposition</Operation>
        <Consequence Type="Improper Address" Comment="">Over
        <Attributes>
            <Operation>
                <Attribute Type="Mechanism">Sequential</Attr
                <Attribute Type="Source Code">Codebase</Attr
                <Attribute Type="Execution Space">Userland</
            </Operation>
            <Operand Name="Address">
                <Attribute Type="Location">Heap</Attribute>
            </Operand>
        </Attributes>
    </Weakness>
```

```
    <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>
    <Failure Type="_XXX" Class="IEX">
        <Cause Type="Memory Error" Comment="">Buffer Overflow</Cause>
        <Operation Comment="">IEX Ooperation</Operation>
        <Consequence Type="Risk" Comment="">IEX Conseqeunce</Consequence>
    </Failure>
</Vulnerability>
```

Previously – Heartbleed (CVE-2014-0160)



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*

2016 IEEE International Conference on Software Quality, Reliability and Security

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

Irena Bojanova
NIST
Gaithersburg, USA
irena.bojanova@nist.gov

Paul E. Black
NIST
Gaithersburg, USA
paul.black@nist.gov

Yaacov Yesha
NIST; UMBC
Gaithersburg, USA; Baltimore, USA
yaacov.yesha@nist.gov

Yan Wu
BGSU
Bowling Green, USA
yanwu@bgsu.edu

***Abstract*—To achieve higher levels of assurance for digital systems, we need to answer questions such as does this software have bugs of these critical classes? Do two software assurance tools find the same set of bugs or different, complimentary sets? Can we guarantee that a new technique discovers all problems of this type? To answer such questions, we need a vastly improved way to describe classes of vulnerabilities and chains of failures. We present the Bugs Framework (BF), which raises the current realm of best efforts and useful heuristics. Our BF includes rigorous definitions and (static) attributes of bug classes, along with their related dynamic properties, such as proximate, secondary and tertiary causes, consequences and sites. The paper discusses the buffer overflow class, the injection class and the control of interaction frequency class, and provides examples of applying our BF taxonomy to describe particular vulnerabilities.**

***Keywords*—software weaknesses; bug taxonomy; attacks.**

I. INTRODUCTION

The medical profession has an extensive, elaborate vocabulary to precisely name muscles, bones, organs and diseases. When a doctor says that a comatose patient has a left temporal lobe epidural hematoma, the intention is to enlighten, not obfuscate. In the software profession, many efforts have developed terms to discuss software, faults, failures and attacks, such as the Common Weakness Enumeration (CWE) [1] and Landwehr et. al. Taxonomy of Computer Program Security Flaws [2], but much work remains.

We want to more accurately and precisely define software bugs or vulnerabilities. Consider that adding "canary" values around arrays detects some buffer overflows while using address layout randomization mitigates others. A precise, orthogonal nomenclature can state exactly which classes of buffer overflows each approach handles. We can also clearly state the classes of bugs that a tool can find and more easily determine if two tools generally find the same set of bugs or if they find different, complimentary sets.

The ancient Greeks used the terms element and atom, and Aristotle proposed that all matter is a mixture of earth, air, fire or water. In the Middle Ages, alchemists made lists of materials, such as alcohol, sulfur, mercury and salt. Through centuries of experimentation and development of scientific principles, we now have Mendeleev's Periodic Table of Elements, see Fig. 1. Just as the structure of the periodic table reflects the underlying atomic structure, we are developing a taxonomy dictated by the "natural" organization of software bugs, while using as stepping stones known bugs enumerations, compendia and collections.

Over the course of history, science has developed many different organizational structures. Linnaeus' taxonomy categorizes living things into a hierarchy of Domain, Kingdom, Phylum, Class, Order, Family, Genus and Species. It allows comprehension of the diversity of life forms and codifies understanding that some animals are close in their evolutionary history. The Geographic Coordinate System specifies any location on Earth using latitude, longitude and elevation. The Dewey Decimal Classification system allows new books and whole new subjects to be placed in reasonable locations in a library for easy retrieval based on subject. Fingerprints are

Fig. 1. Periodic Table of Elements: 🟥 antiquity, 🟧 Levoisier 1789, 🟨 Mendeleev 1869, 🟩 Deming 1923, 🟦 Seaborg 1945, ⬜ up to 2000, 🟪 to 2012[1].

---

[1] By Sandbh - Wikimedia Commons., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=31017351

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.29

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 Weaknesse
Reintroducing Common Weaknes

Yan Wu, Bowling Green State University
Irena Bojanova, University of Maryland, Baltimore County
Yaacov Yesha, University of Maryland University College

Abstract: Knowing what makes your software systems vulnerable to attacks is critical, as software vulnerabilities hurt security, reliability, and availability of the system as a whole. The Common Weakness Enumeration (CWE), a community effort that provides the foundation for such knowledge, is not sufficient, accurate and precise enough to serve as the common language measuring stick and provide a common baseline for developers and security practitioners. In this article, we introduce the relevant body of knowledge that consolidates CWE, including the Semantic Template and Software Fault Pattern efforts, and how static analysis tools add value through CWEs. We also provide future directions, present our vision on CWE formalization, and discuss the value of CWE for not only software assurance community, but also for Computer Science.

1. Introduction to Common Weakness Enumeration (CWE)

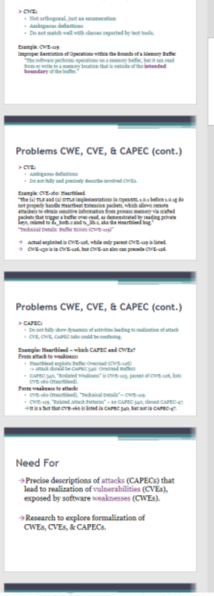
Software weaknesses could be exploited to compromise a system's security. This is especially critical for systems such as the Department of Defense (DoD) systems, in which the amount of software is very large. Software assurance countermeasures should be applied to address anticipated attacks against a system. Such attacks are enabled by software vulnerabilities, and those countermeasures reduce those vulnerabilities or remove them[12].

Common Weakness Enumeration (CWE) [1] is a collection of software weakness descriptions that offers a way to identify and eliminate vulnerabilities in computer systems. CWE is also used to evaluate the tools and services developed for finding weaknesses in software. CWE is community-developed and maintained by MITRE Corporation [1].

A preliminary classification of vulnerabilities, attacks, and related concepts was developed by MITRE's CVE [2] team. That effort began in 2005., CWE was developed as a list of software weaknesses that is more suitable for software security assessment [14].

by answering three basic questions:

44 CrossTalk—September/October 2015



Problems with CWE, CVE, & CAPEC

- CWE:
  - Not orthogonal, just an enumeration
  - Ambiguous definitions
  - Do not match well with classes reported by test tools.

Example: CWE-119:
Improper Restriction of Operations within the Bounds of a Memory Buffer
"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."

Problems CWE, CVE, & CAPEC (cont.)

- CVE:
  - Ambiguous definitions
  - Do not fully and precisely describe involved CWEs.

Example: CVE-160: 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."
"Technical Details: Buffer Errors (CWE-119)"

→ Actual exploited is CWE-126, while only parent CWE-119 is listed.
→ CWE-130 is in CWE-126, but CWE-20 also can precede CWE-126.

Problems CWE, CVE, & CAPEC (cont.)

- CAPEC:
  - Do not fully show dynamics of activities leading to realization of attack
  - CVE, CWE, CAPEC inks could be confusing.

Example: Heartbleed – which CAPEC and CWEs?
From attack to weakness:
- Heartbleed exploits Buffer Overread (CWE-126) -> attack should be CAPEC 540: Overread Buffers
- CAPEC 540, "Related Weakness" is CWE-125, parent of CWE-126, lists CVE-160 (Heartbleed).

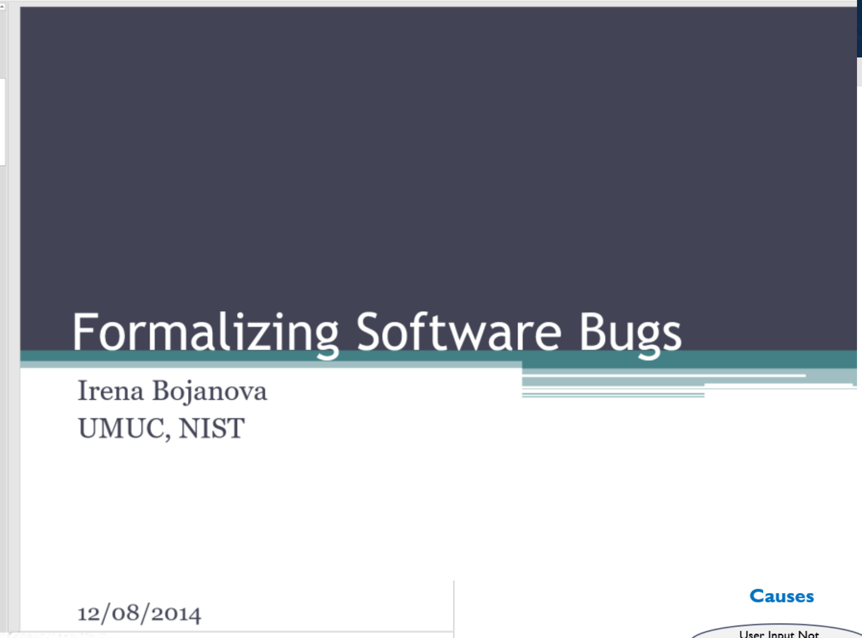
Form weakness to attack:
- CVE-160 (Heartbleed), "Technical Details" – CWE-119.
- CWE-119, "Related Attack Patterns" – no CAPEC 540, direct CAPEC-47.

→ It is a fact that CVE-160 is listed in CAPEC 540, but not in CAPEC-47.

Need For

→ Precise descriptions of attacks (CAPECs) that lead to realization of vulnerabilities (CVEs), exposed by software weaknesses (CWEs).

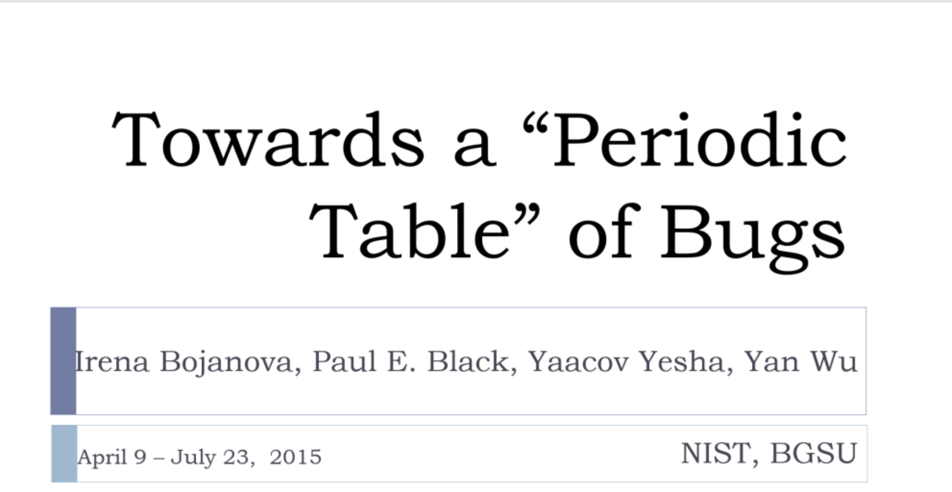
→ Research to explore formalization of CWEs, CVEs, & CAPECs.



Formalizing Software Bugs

Irena Bojanova
UMUC, NIST

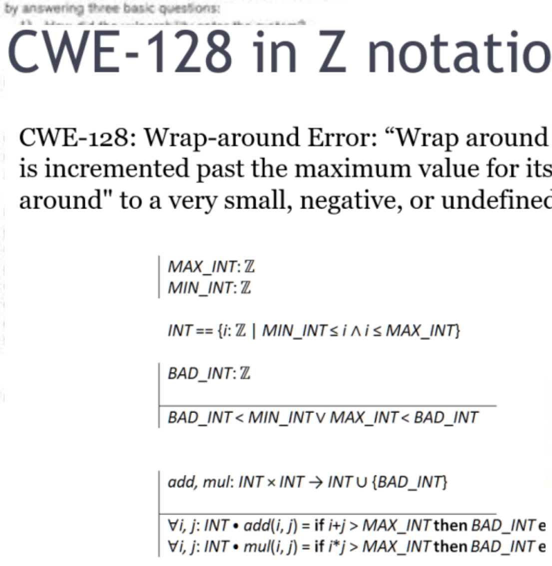
12/08/2014



Towards a "Periodic Table" of Bugs

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

April 9 – July 23, 2015 NIST, BGSU



CWE-128 in Z notation

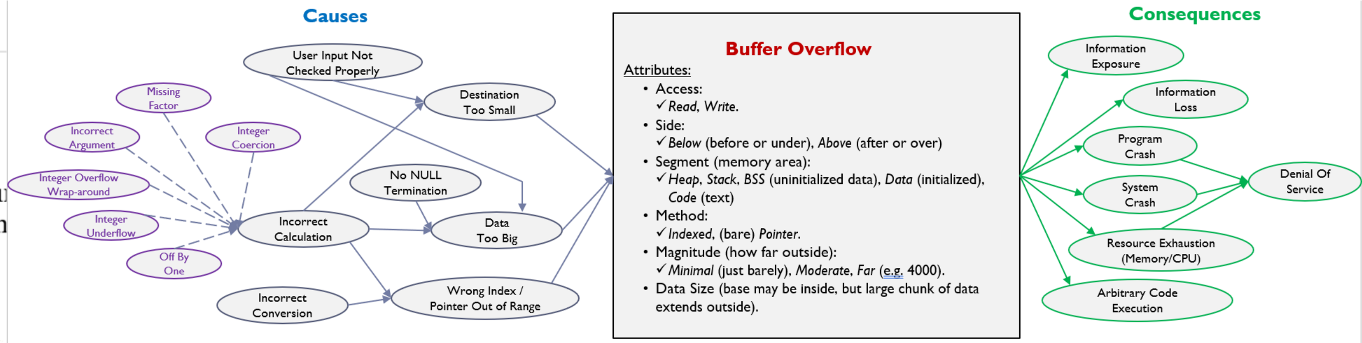
CWE-128: Wrap-around Error: "Wrap around errors occu
is incremented past the maximum value for its type and th
around" to a very small, negative, or undefined value."

$$MAX\_INT : \mathbb{Z}$$
$$MIN\_INT : \mathbb{Z}$$

$$INT == \{i : \mathbb{Z} \mid MIN\_INT \le i \land i \le MAX\_INT\}$$

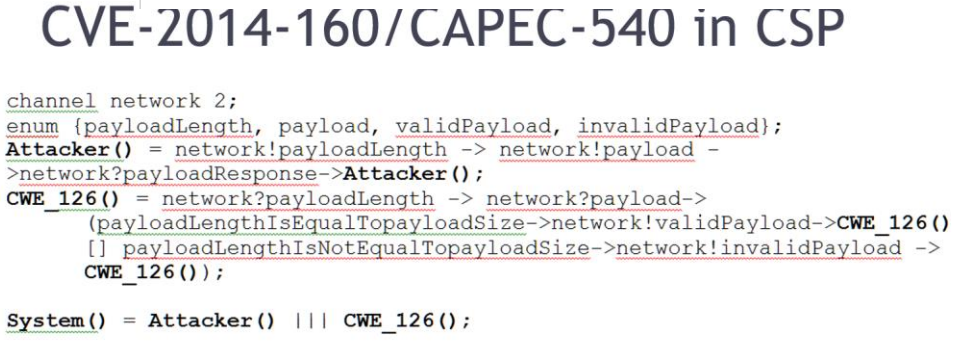
$$BAD\_INT : \mathbb{Z}$$
$$BAD\_INT < MIN\_INT \lor MAX\_INT < BAD\_INT$$

$$add, mul : INT \times INT \rightarrow INT \cup \{BAD\_INT\}$$
$$\forall i, j : INT \bullet add(i, j) = \textbf{if } i+j > MAX\_INT \textbf{ then } BAD\_INT \textbf{ e}$$
$$\forall i, j : INT \bullet mul(i, j) = \textbf{if } i*j > MAX\_INT \textbf{ then } BAD\_INT \textbf{ e}$$



Buffer Overflow

Attributes:
- Access:
  - ✓ *Read*, *Write*.
- Side:
  - ✓ *Below* (before or under), *Above* (after or over)
- Segment (memory area):
  - ✓ *Heap*, *Stack*, *BSS* (uninitialized data), *Data* (initialized), *Code* (text)
- Method:
  - ✓ *Indexed*, (bare) *Pointer*.
- Magnitude (how far outside):
  - ✓ *Minimal* (just barely), *Moderate*, *Far* (e.g. 4000).
- Data Size (base may be inside, but large chunk of data extends outside).



CVE-2014-160/CAPEC-540 in CSP

```
channel network 2;
enum {payloadLength, payload, validPayload, invalidPayload};
Attacker() = network!payloadLength -> network!payload -
>network?payloadResponse->Attacker();
CWE_126() = network?payloadLength -> network?payload->
      (payloadLengthIsEqualTopayloadSize->network!validPayload->CWE_126()
      [] payloadLengthIsNotEqualTopayloadSize->network!invalidPayload ->
      CWE_126());

System() = Attacker() ||| CWE_126();
```

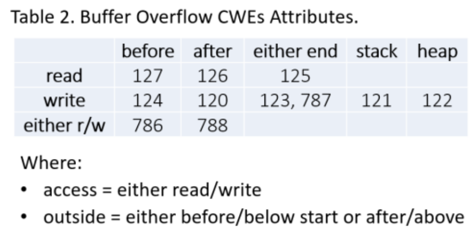
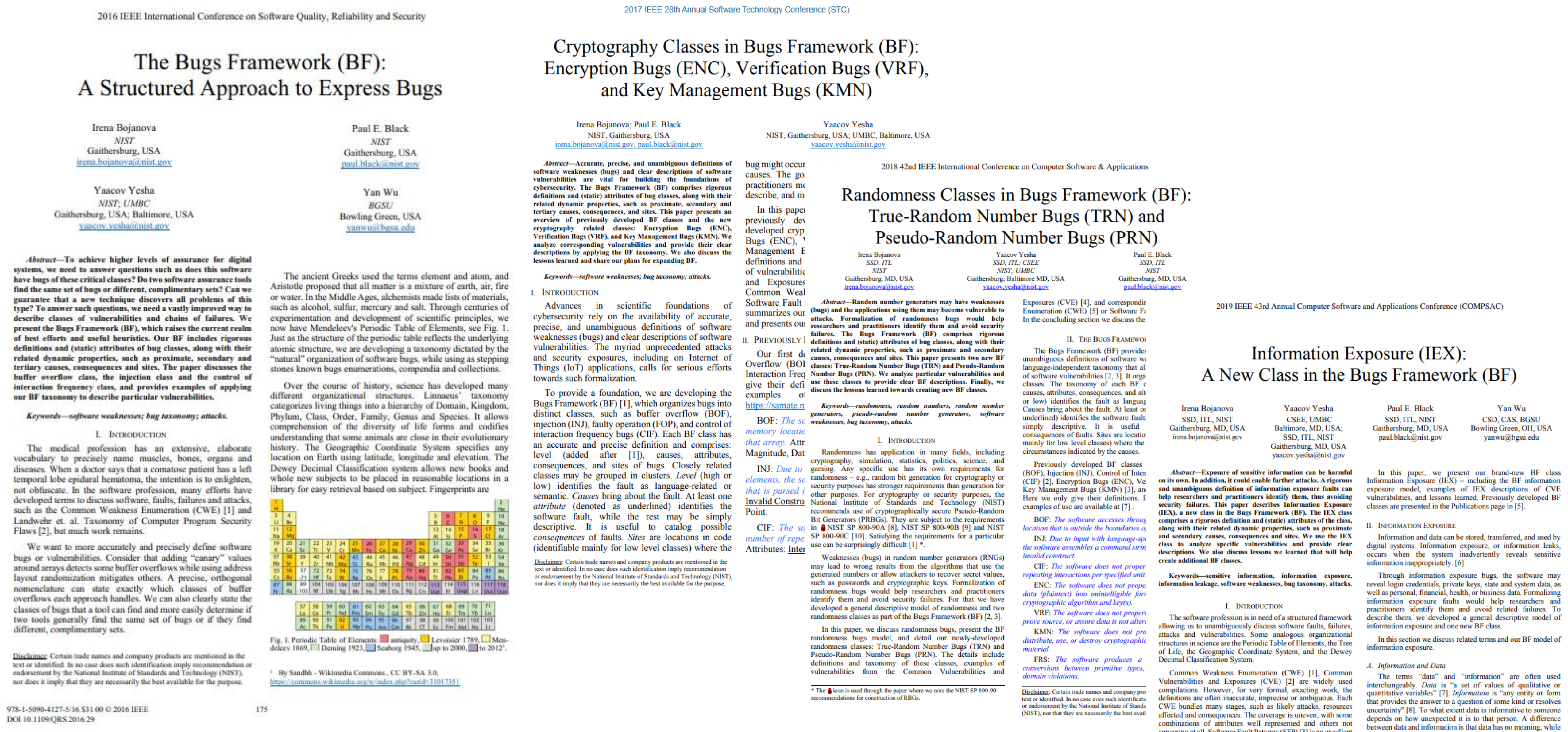


Table 2. Buffer Overflow CWEs Attributes.

	before	after	either end	stack	heap
read	127	126	125		
write	124	120	123, 787	121	122
either r/w	786	788			

Where:
- access = either read/write
- outside = either before/below start or after/above

Next BF Classes



2016 IEEE International Conference on Software Quality, Reliability and Security

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

Irena Bojanova
NIST
Gaithersburg, USA
irena.bojanova@nist.gov

Paul E. Black
NIST
Gaithersburg, USA
paul.black@nist.gov

Yaacov Yesha
NIST; UMBC
Gaithersburg, USA; Baltimore, USA
yaacov.yesha@nist.gov

Yan Wu
BGSU
Bowling Green, USA
yanwu@bgsu.edu

Abstract—**To achieve higher levels of assurance for digital systems, we need to answer questions such as does this software have bugs of these critical classes? Do two software assurance tools find the same set of bugs or different, complimentary sets? Can we guarantee that a new technique discovers all problems of this type? To answer such questions, we need a vastly improved way to describe classes of vulnerabilities and chains of failures. We present the Bugs Framework (BF), which raises the current realm of best efforts and useful heuristics. Our BF includes rigorous definitions and (static) attributes of bug classes, along with their related dynamic properties, such as proximate, secondary and tertiary causes, consequences and sites. The paper discusses the buffer overflow class, the injection class and the control of interaction frequency class, and provides examples of applying our BF taxonomy to describe particular vulnerabilities.**

Keywords—software weaknesses; bug taxonomy; attacks.

I. INTRODUCTION

The medical profession has an extensive, elaborate vocabulary to precisely name muscles, bones, organs and diseases. When a doctor says that a comatose patient has a left temporal lobe epidural hematoma, the intention is to enlighten, not obfuscate. In the software profession, many efforts have developed terms to discuss software, faults, failures and attacks, such as the Common Weakness Enumeration (CWE) [1] and Landwehr et. al. Taxonomy of Computer Program Security Flaws [2], but much work remains.

We want to more accurately and precisely define software bugs or vulnerabilities. Consider that adding "canary" values around arrays detects some buffer overflows while using address layout randomization mitigates others. A precise, orthogonal nomenclature can state exactly which classes of buffer overflows each approach handles. We can also clearly state the classes of bugs that a tool can find and more easily determine if two tools generally find the same set of bugs or if they find different, complimentary sets.

The ancient Greeks used the terms element and atom, and Aristotle proposed that all matter is a mixture of earth, air, fire or water. In the Middle Ages, alchemists made lists of materials, such as alcohol, sulfur, mercury and salt. Through centuries of experimentation and development of scientific principles, we now have Mendeleev's Periodic Table of Elements, see Fig. 1. Just as the structure of the periodic table reflects the underlying atomic structure, we are developing a taxonomy dictated by the "natural" organization of software bugs, while using as stepping stones known bugs enumerations, compendia and collections.

Over the course of history, science has developed many different organizational structures. Linnaeus' taxonomy categorizes living things into a hierarchy of Domain, Kingdom, Phylum, Class, Order, Family, Genus and Species. It allows comprehension of the diversity of life forms and codifies understanding that some animals are close in their evolutionary history. The Geographic Coordinate System specifies any location on Earth using latitude, longitude and elevation. The Dewey Decimal Classification system allows new books and whole new subjects to be placed in reasonable locations in a library for easy retrieval based on subject. Fingerprints are

Fig. 1. Periodic Table of Elements[1]: 🟥 antiquity, 🟧 Lavoisier 1789, 🟨 Mendeleev 1869, 🟩 Deming 1923, 🟦 Seaborg 1945, ☐ up to 2000, 🟪 to 2012.

---

[1] By Sandbh - Wikimedia Commons., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=31017351

978-1-5090-4127-5/16 $31.00 © 2016 IEEE 175
DOI 10.1109/QRS.2016.29

2017 IEEE 28th Annual Software Technology Conference (STC)

Cryptography Classes in Bugs Framework (BF): Encryption Bugs (ENC), Verification Bugs (VRF), and Key Management Bugs (KMN)

Irena Bojanova; Paul E. Black
NIST, Gaithersburg, USA
irena.bojanova@nist.gov, paul.black@nist.gov

Yaacov Yesha
NIST, Gaithersburg, USA; UMBC, Baltimore, USA
yaacov.yesha@nist.gov

Abstract—**Accurate, precise, and unambiguous definitions of software weaknesses (bugs) and clear descriptions of software vulnerabilities are vital for building the foundations of cybersecurity. The Bugs Framework (BF) comprises rigorous definitions and (static) attributes of bug classes, along with their related dynamic properties, such as proximate, secondary and tertiary causes, consequences, and sites. This paper presents an overview of previously developed BF classes and the new cryptography related classes: Encryption Bugs (ENC), Verification Bugs (VRF), and Key Management Bugs (KMN). We analyze corresponding vulnerabilities and provide their clear descriptions by applying the BF taxonomy. We also discuss the lessons learned and share our plans for expanding BF.**

Keywords—software weaknesses; bug taxonomy; attacks.

I. INTRODUCTION

Advances in scientific foundations of cybersecurity rely on the availability of accurate, precise, and unambiguous definitions of software weaknesses (bugs) and clear descriptions of software vulnerabilities. The myriad unprecedented attacks and security exposures, including on Internet of Things (IoT) applications, calls for serious efforts towards such formalization.

To provide a foundation, we are developing the Bugs Framework (BF) [1], which organizes bugs into distinct classes, such as buffer overflow (BOF), injection (INJ), faulty operation (FOP), and control of interaction frequency bugs (CIF). Each BF class has an accurate and precise definition and comprises: level (added after [1]), causes, attributes, consequences, and sites of bugs. Closely related classes may be grouped in clusters. *Level* (high or low) identifies the fault as language-related or semantic. *Causes* bring about the fault. At least one *attribute* (denoted as underlined) identifies the software fault, while the rest may be simply descriptive. It is useful to catalog possible *consequences* of faults. *Sites* are locations in code (identifiable mainly for low level classes) where the

bug might occur
causes. The goa
practitioners mo
describe, and m

In this paper
previously dev
developed cryp
Bugs (ENC),
Management B
definitions and
of vulnerabilitie
and Exposures
Common Weak
Software Fault
summarizes our
and presents ou

II. PREVIOUSLY

Our first d
Overflow (BOF
Interaction Freq
give their defi
examples of
https://samate.n

BOF: *The so*
memory locatio
that array. Attr
Magnitude, Dat

INJ: *Due to*
elements, the so
that is parsed i
Invalid Constru
Point.

CIF: *The so*
number of repe
Attributes: Inter

2018 42nd IEEE International Conference on Computer Software & Applications

Randomness Classes in Bugs Framework (BF): True-Random Number Bugs (TRN) and Pseudo-Random Number Bugs (PRN)

Irena Bojanova
SSD, ITL
NIST
Gaithersburg, MD, USA
irena.bojanova@nist.gov

Yaacov Yesha
SSD, ITL; CSEE
NIST; UMBC
Gaithersburg; Baltimore MD, USA
yaacov.yesha@nist.gov

Paul E. Black
SSD, ITL
NIST
Gaithersburg, MD, USA
paul.black@nist.gov

Abstract—**Random number generators may have weaknesses (bugs) and the applications using them may become vulnerable to attacks. Formalization of randomness bugs would help researchers and practitioners identify them and avoid security failures. The Bugs Framework (BF) comprises rigorous definitions and (static) attributes of bug classes, along with their related dynamic properties, such as proximate and secondary causes, consequences and sites. This paper presents two new BF classes: True-Random Number Bugs (TRN) and Pseudo-Random Number Bugs (PRN). We analyze particular vulnerabilities and use these classes to provide clear BF descriptions. Finally, we discuss the lessons learned towards creating new BF classes.**

Keywords—randomness, random numbers, random number generators, pseudo-random number generators, software weaknesses, bug taxonomy, attacks.

I. INTRODUCTION

Randomness has application in many fields, including cryptography, simulation, statistics, politics, science, and gaming. Any specific use has its own requirements for randomness – e.g., random bit generation for cryptography or security purposes has stronger requirements than generation for other purposes. For cryptography or security purposes, the National Institute of Standards and Technology (NIST) recommends use of cryptographically secure Pseudo-Random Bit Generators (PRBGs). They are subject to the requirements in NIST SP 800-90A [8], NIST SP 800-90B [9] and NIST SP 800-90C [10]. Satisfying the requirements for a particular use can be surprisingly difficult [1] *.

Weaknesses (bugs) in random number generators (RNGs) may lead to wrong results from the algorithms that use the generated numbers or allow attackers to recover secret values, such as passwords and cryptographic keys. Formalization of randomness bugs would help researchers and practitioners identify them and avoid security failures. For that we have developed a general descriptive model of randomness and two randomness classes as part of the Bugs Framework (BF) [2, 3].

In this paper, we discuss randomness bugs, present the BF randomness bugs model, and detail our newly-developed randomness classes: True-Random Number Bugs (TRN) and Pseudo-Random Number Bugs (PRN). The details include definitions and taxonomy of these classes, examples of vulnerabilities from the Common Vulnerabilities and Exposures (CVE) [4], and correspondin
Enumeration (CWE) [5] or Software Fa
In the concluding section we discuss the

II. THE BUGS FRAMEWO

The Bugs Framework (BF) provides
unambiguous definitions of software w
language-independent taxonomy that al
of software vulnerabilities [2, 3]. It orga
classes. The taxonomy of each BF c
causes, attributes, consequences, and sit
or low) identifies the fault as languag
Causes bring about the fault. At least o
underlined) identifies the software fault,
simply descriptive. It is useful
consequences of faults. Sites are locatio
mainly for low level classes) where the
circumstances indicated by the causes.

Previously developed BF classes
(BOF), Injection (INJ), Control of Inter
(CIF) [2], Encryption Bugs (ENC), Ve
Key Management Bugs (KMN) [3], an
Here we only give their definitions. I
examples of use are available at [7].

BOF: *The software accesses throug*
location that is outside the boundaries o

INJ: *Due to input with language-sp*
the software assembles a command strin
invalid construct.

CIF: *The software does not proper*
repeating interactions per specified unit.

ENC: *The software does not prope*
data (plaintext) into unintelligible for
cryptographic algorithm and key(s).

VRF: *The software does not proper*
prove source, or assure data is not alter

KMN: *The software does not pro*
distribute, use, or destroy cryptographic
material.

FRS: *The software produces a*
conversions between primitive types,
domain violations.

---

* The 🔴 icon is used through the paper where we note the NIST SP 800-90 recommendations for construction of RBGs.

2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Information Exposure (IEX): A New Class in the Bugs Framework (BF)

Irena Bojanova
SSD, ITL, NIST
Gaithersburg, MD, USA
irena.bojanova@nist.gov

Yaacov Yesha
CSEE, UMBC
Baltimore, MD, USA;
SSD, ITL, NIST
Gaithersburg, MD, USA
yaacov.yesha@nist.gov

Paul E. Black
SSD, ITL, NIST
Gaithersburg, MD, USA
paul.black@nist.gov

Yan Wu
CSD, CAS, BGSU
Bowling Green, OH, USA
yanwu@bgsu.edu

Abstract—**Exposure of sensitive information can be harmful on its own. In addition, it could enable further attacks. A rigorous and unambiguous definition of information exposure faults can help researchers and practitioners identify them, thus avoiding security failures. This paper describes Information Exposure (IEX), a new class in the Bugs Framework (BF). The IEX class comprises a rigorous definition and (static) attributes of the class, along with their related dynamic properties, such as proximate and secondary causes, consequences and sites. We use the IEX class to analyze specific vulnerabilities and provide clear descriptions. We also discuss lessons we learned that will help create additional BF classes.**

Keywords—sensitive information, information exposure, information leakage, software weaknesses, bug taxonomy, attacks.

I. INTRODUCTION

The software profession is in need of a structured framework allowing us to unambiguously discuss software faults, failures, attacks and vulnerabilities. Some analogous organizational structures in science are the Periodic Table of Elements, the Tree of Life, the Geographic Coordinate System, and the Dewey Decimal Classification System.

Common Weakness Enumeration (CWE) [1], Common Vulnerabilities and Exposures (CVE) [2] are widely used compilations. However, for very formal, exacting work, the definitions are often inaccurate, imprecise or ambiguous. Each CWE bundles many stages, such as likely attacks, resources affected and consequences. The coverage is uneven, with some combinations of attributes well represented and others not appearing at all. Software Fault Patterns (SFP) [3] is an excellent

In this paper, we present our brand-new BF class Information Exposure (IEX) – including the BF information exposure model, examples of IEX descriptions of CVE vulnerabilities, and lessons learned. Previously developed BF classes are presented in the Publications page in [5].

II. INFORMATION EXPOSURE

Information and data can be stored, transferred, and used by digital systems. Information exposure, or information leaks, occurs when the system inadvertently reveals sensitive information inappropriately. [6]

Through information exposure bugs, the software may reveal login credentials, private keys, state and system data, as well as personal, financial, health, or business data. Formalizing information exposure faults would help researchers and practitioners identify them and avoid related failures. To describe them, we developed a general descriptive model of information exposure and one new BF class.

In this section we discuss related terms and our BF model of information exposure.

A. Information and Data

The terms "data" and "information" are often used interchangeably. *Data* is "a set of values of qualitative or quantitative variables" [7]. *Information* is "any entity or form that provides the answer to a question of some kind or resolves uncertainty" [8]. To what extent data is informative to someone depends on how unexpected it is to that person. A difference between data and information is that data has no meaning, while

Missing Cornerstones

- Strict Definitions of:
    - Bug
    - Weakness
    - Vulnerability
    - Failure

- Clarity on:
    - Chaining Bugs/Weaknesses/Failures
    - Merging Chains

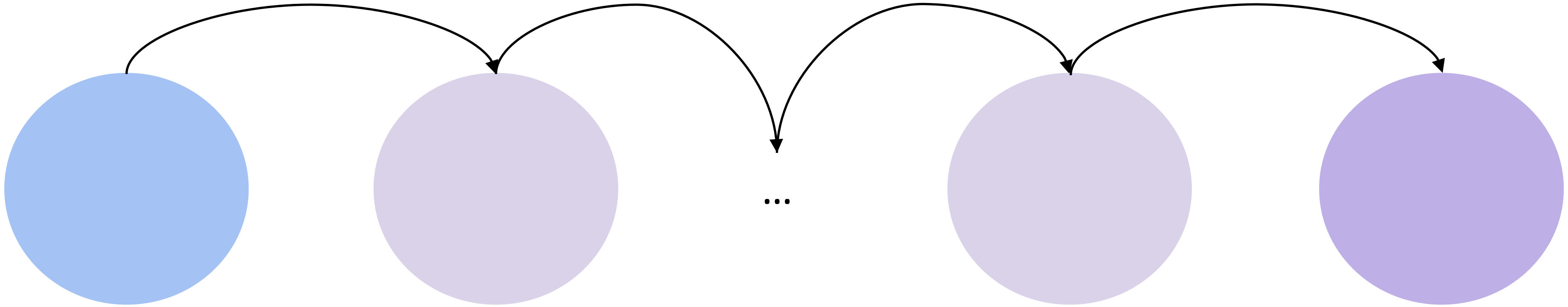
Terminology

- 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

I. Bojanova and C. Eduardo Galhardo, "Classifying Memory Bugs Using Bugs Framework Approach," 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 2021, pp. 1157-1164, https://doi.org/10.1109/COMPSAC51774.2021.00159.

BF Goals

1. Solve the problems of imprecise descriptions and unclear causality



2. Solve the problems of gaps and overlaps in coverage

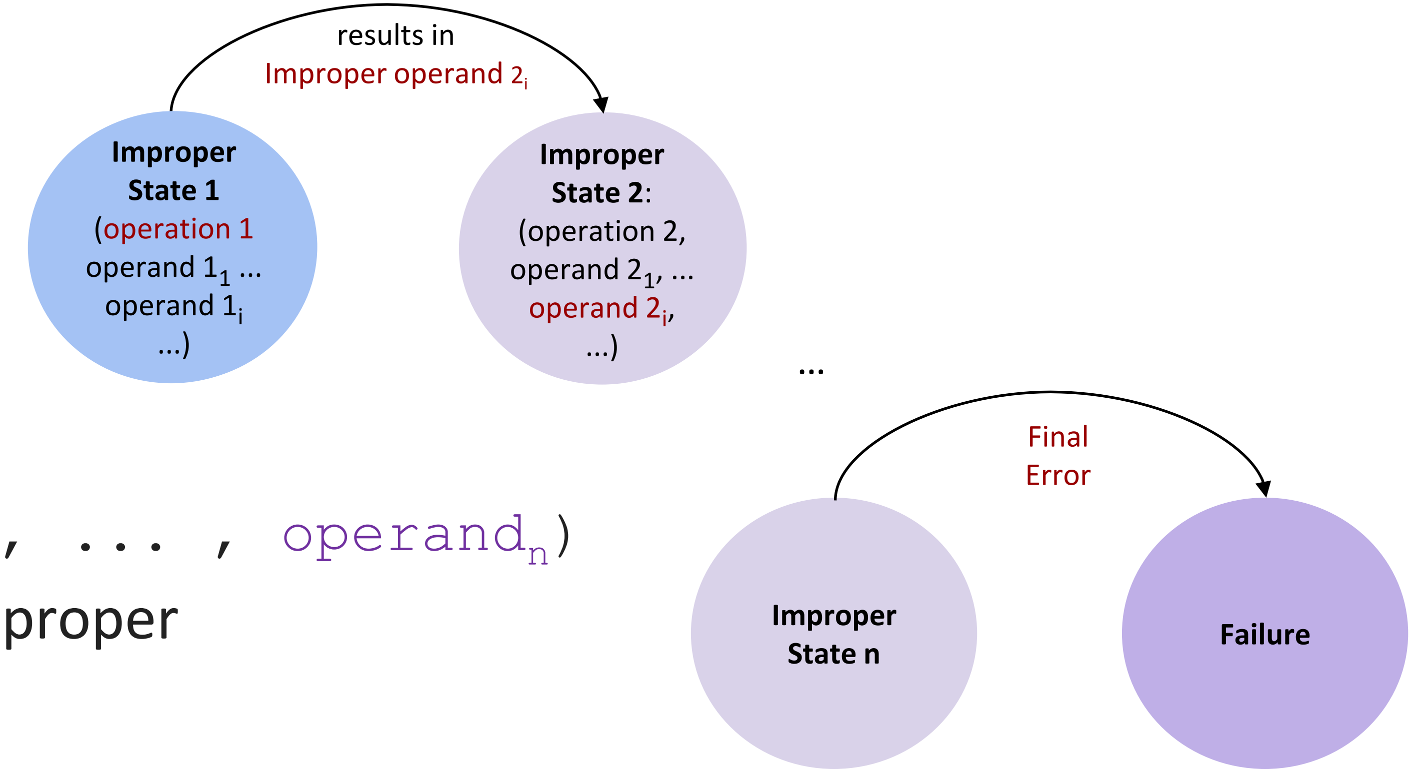
BF Features – Clear Causal Descriptions

- BF describes a bug/weakness as:
  - An improper state and
  - Its transition
- Improper State –

  a tuple $(\mathtt{operation}, \mathtt{operand}_1, \ldots, \mathtt{operand}_n)$

  , where at least one element is improper
- Transition –

  the result of the operation over the operands



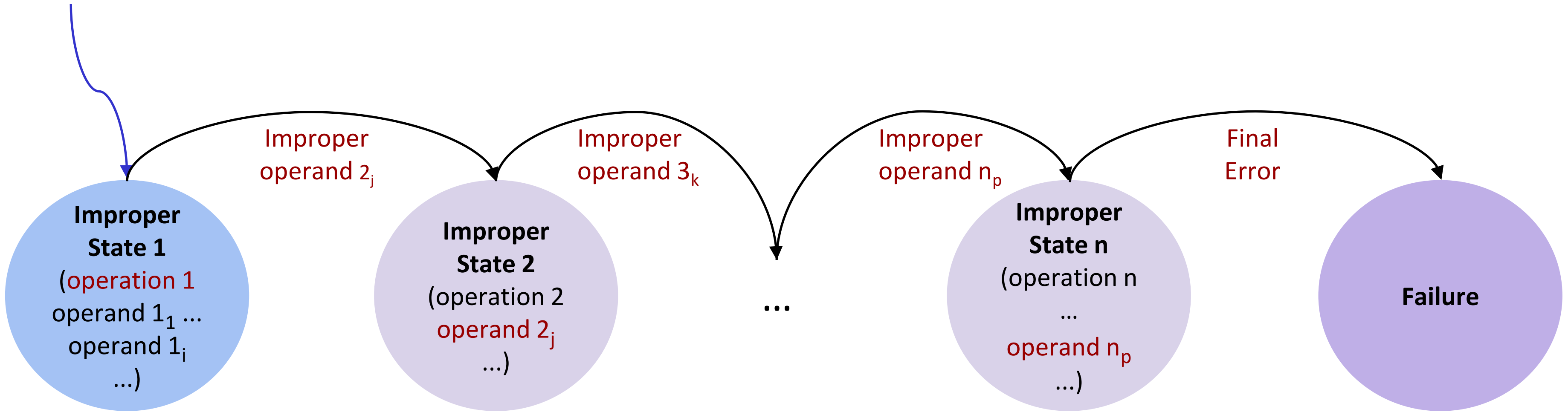
● Initial State – caused by the Bug – the operation is improper

● Intermediate State – caused by ill-formed data – at least one operand is improper

● Final State – the Failure – caused by a final error

BF Features – Chaining Weaknesses

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



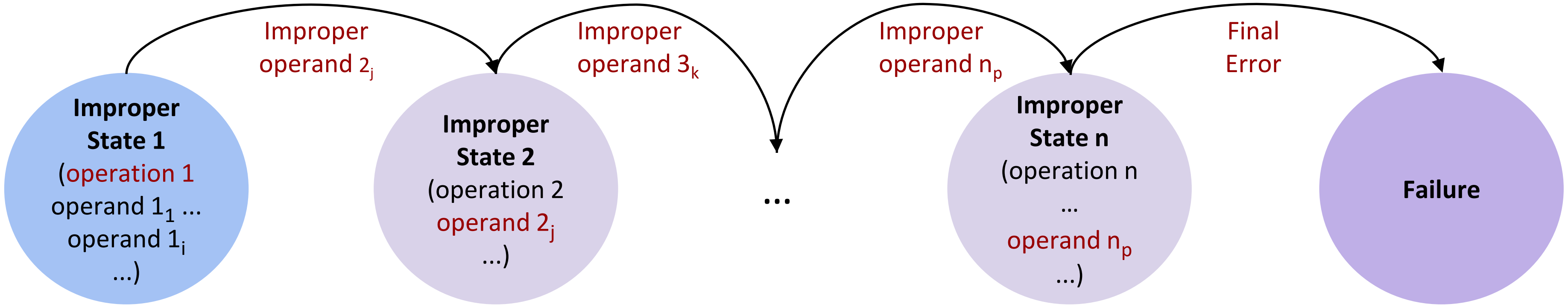
● Initial State – caused by the Bug – the operation is improper

● Intermediate State – caused by ill-formed data – at least one operand is improper

● Final State – the Failure – caused by a final error

BF Features – Backtracking

- How to find the Bug?
- Go backwards by operand until an operation is a cause

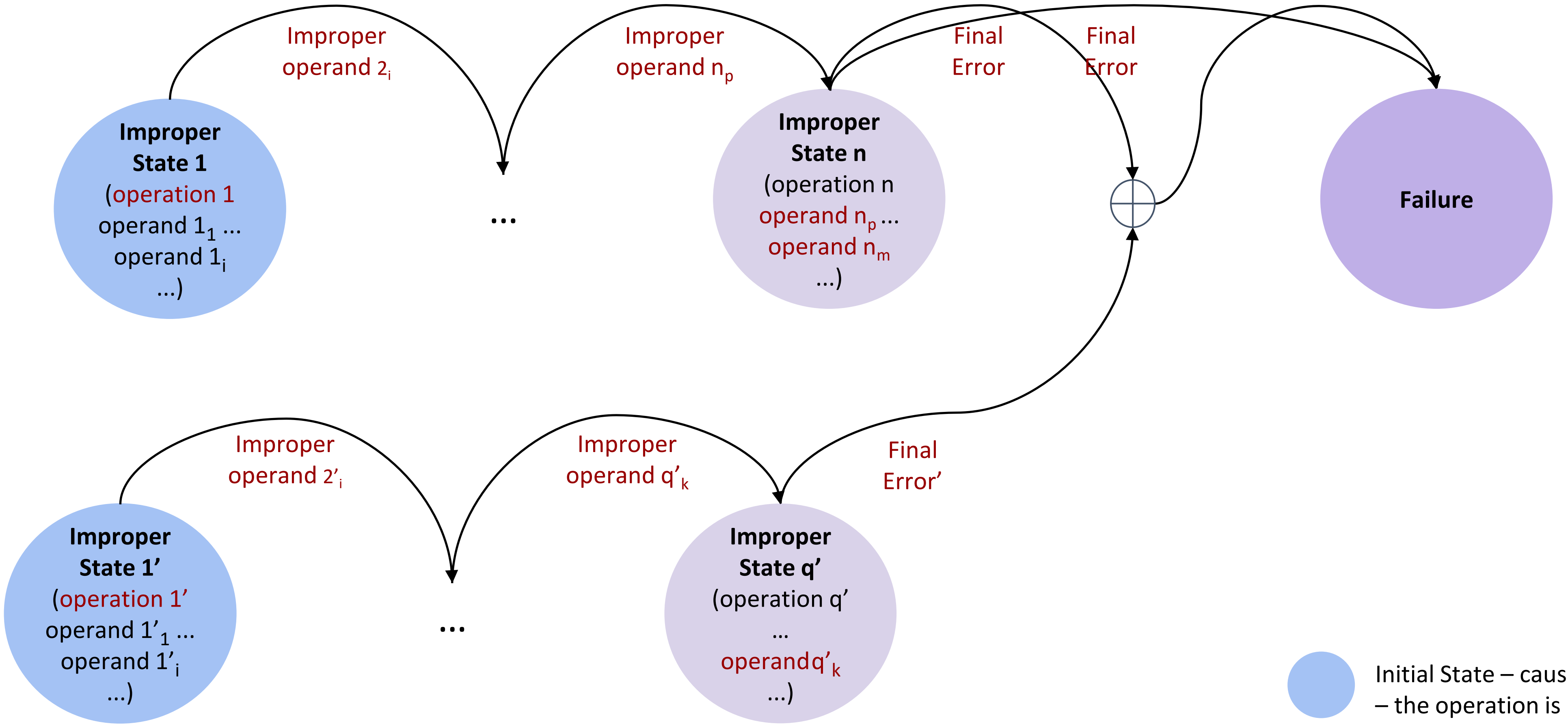


● Initial State – caused by the Bug – the operation is improper

● Intermediate State – caused by ill-formed data – at least one operand is improper

● Final State – the Failure – caused by a final error

BF Features – Converging Vulnerabilities



Improper State 1
(operation 1 operand 1_1 … operand 1_i …)

Improper operand 2_i

…

Improper operand n_p

Improper State n
(operation n operand n_p … operand n_m …)

Final Error

Final Error

Failure

Improper State 1’
(operation 1’ operand $1’_1$ … operand $1’_i$ …)

Improper operand $2’_i$

…

Improper operand $q’_k$

Improper State q’
(operation q’ … operandq’$_k$ …)

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

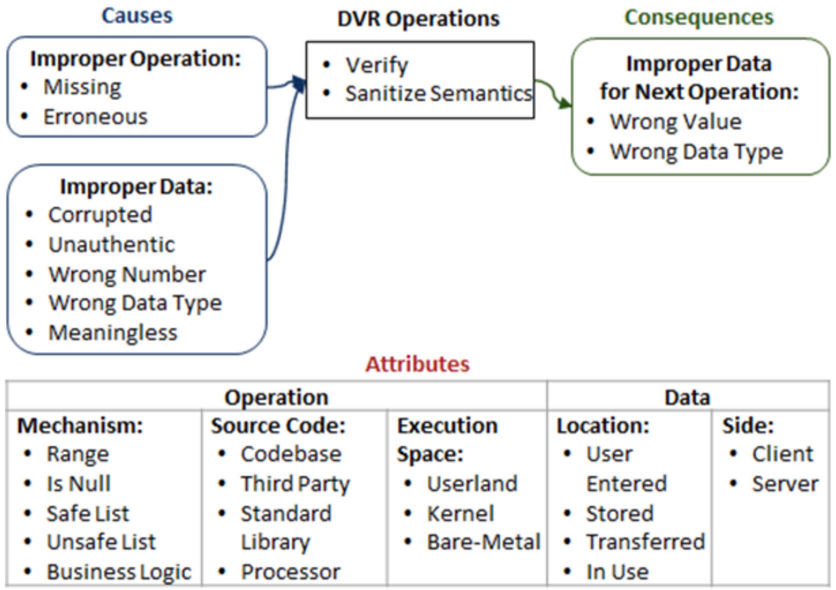
● Final State – the Failure – caused by a final error

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



Improper Operation:
- Missing
- Erroneous

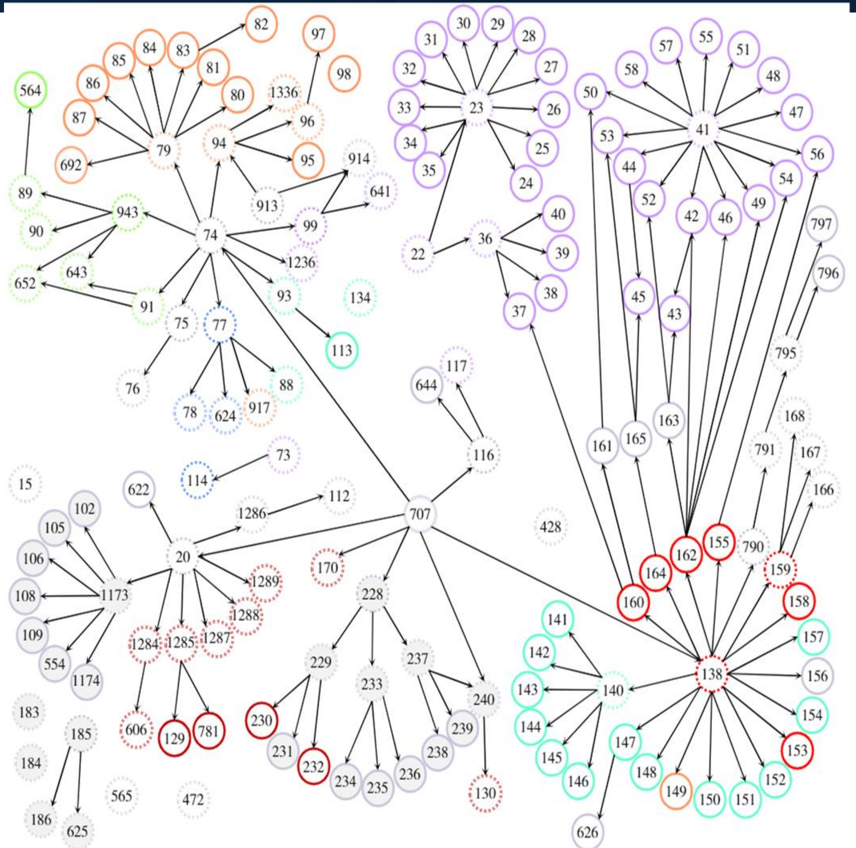
Improper Data:
- Corrupted
- Unauthentic
- Wrong Number
- Wrong Data Type
- Meaningless

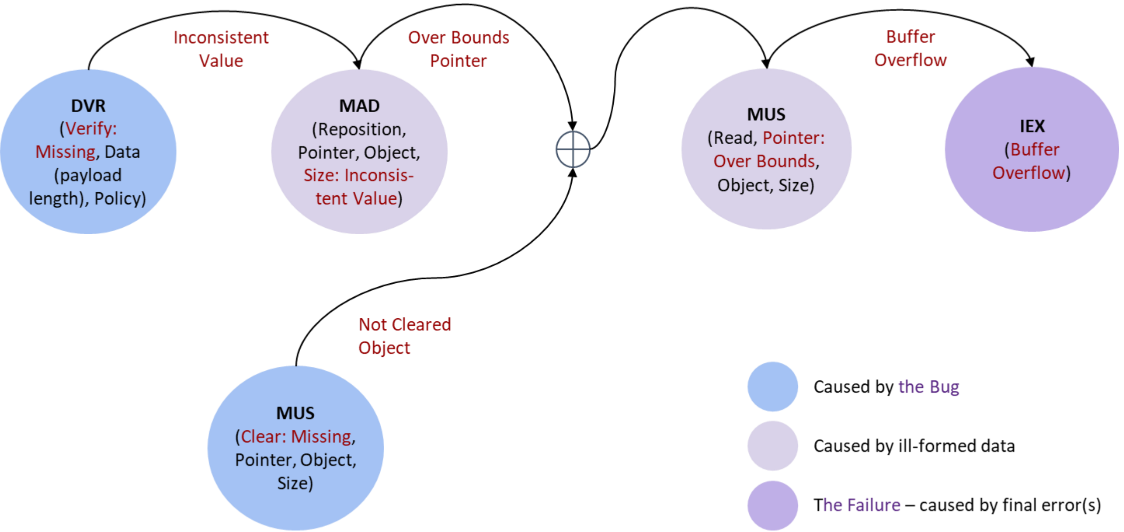
- Verify
- Sanitize Semantics

Improper Data for Next Operation:
- Wrong Value
- Wrong Data Type

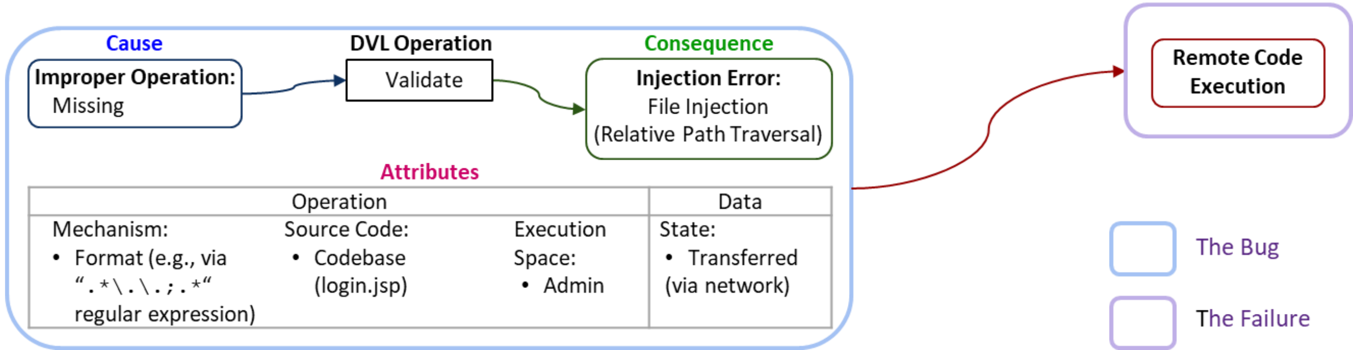
Attributes

Operation			Data	
Mechanism:	**Source Code:**	**Execution Space:**	**Location:**	**Side:**
• Range • Is Null • Safe List • Unsafe List • Business Logic	• Codebase • Third Party • Standard Library • Processor	• Userland • Kernel • Bare-Metal	• User Entered • Stored • Transferred • In Use	• Client • Server





- Caused by the Bug
- Caused by ill-formed data
- The Failure – caused by final error(s)



Improper Operation: Missing

Validate

Injection Error: File Injection (Relative Path Traversal)

Attributes

Operation			Data
Mechanism: • Format (e.g., via “.*\.\.;.*“ regular expression)	Source Code: • Codebase (login.jsp)	Execution Space: • Admin	State: • Transferred (via network)

- The Bug
- The Failure

BF – Defined

- BF is a …
    - Structured
    - Complete
    - Orthogonal
    - Language independent

classification of software bugs and weaknesses

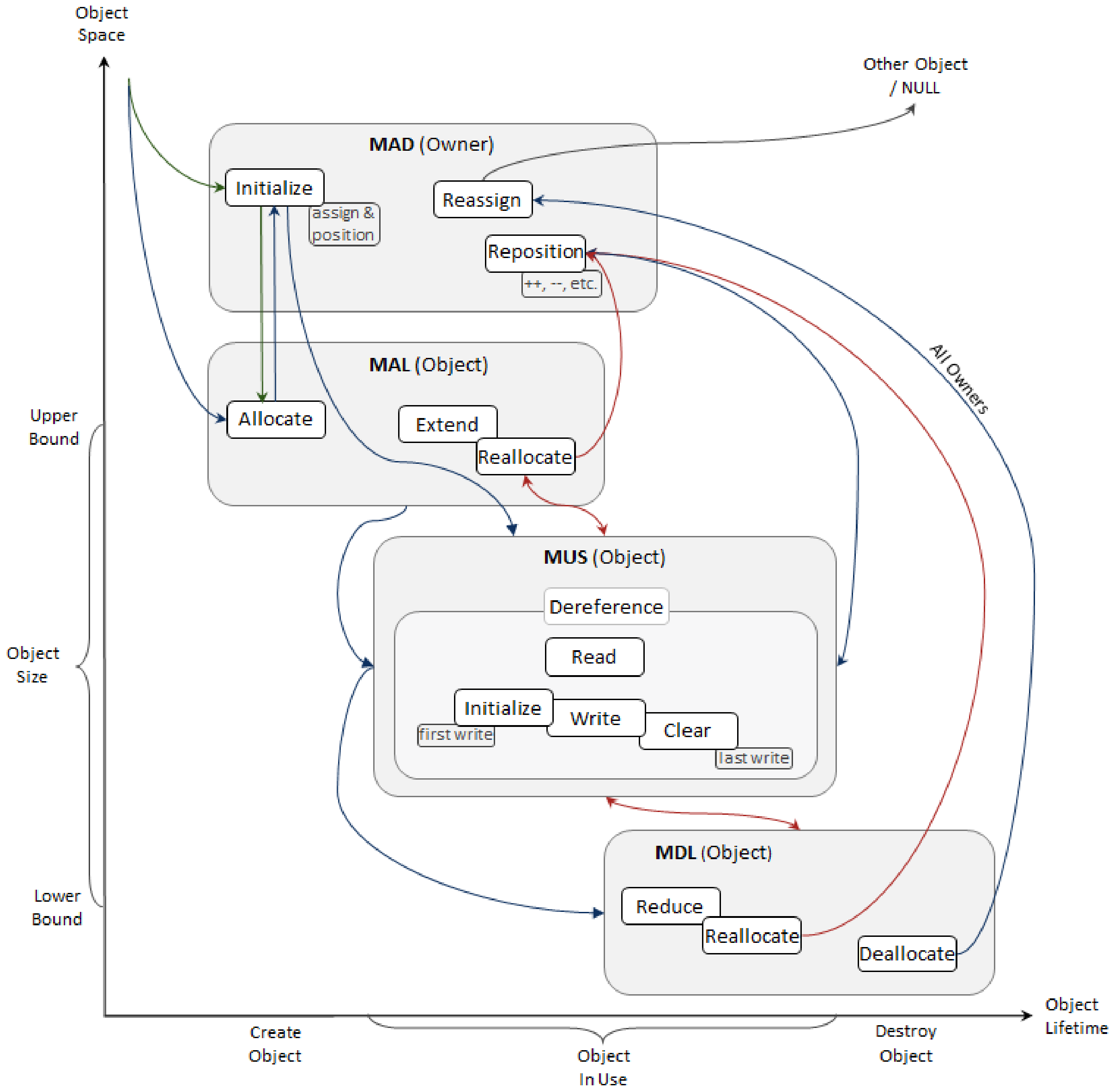
BF – Bugs Models

- Example:

  The BF Memory Bugs Model:

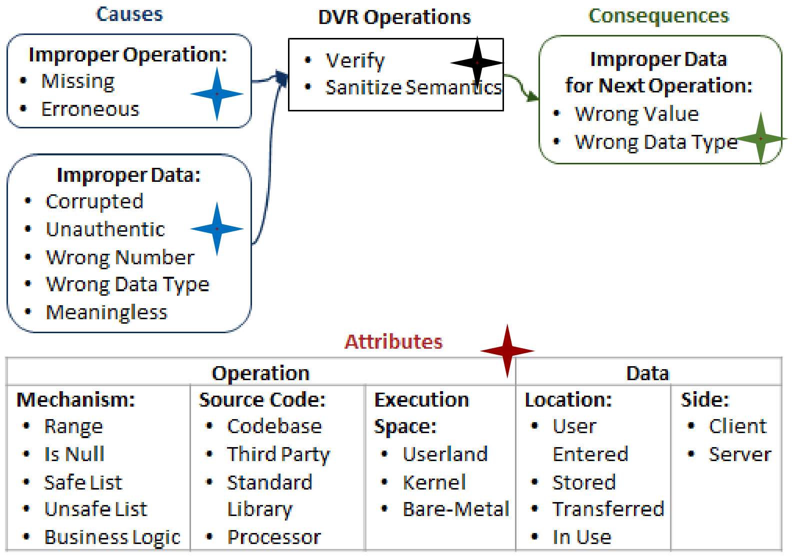
  - Four phases, corresponding to the BF memory bugs classes: MAD, MAL, MUS, MDL

  - 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

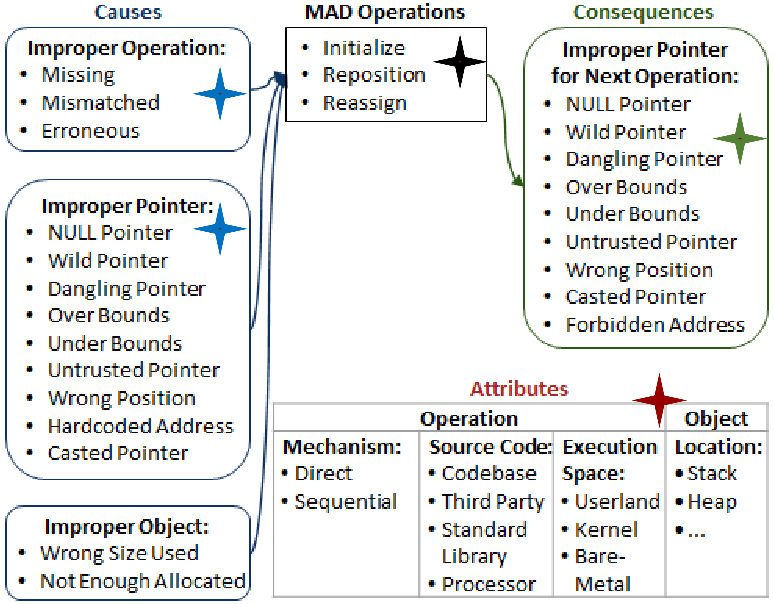
Consequences

Improper Data for Next Operation:
- Wrong Value
- Wrong Data Type

Attributes

Operation			Data	
Mechanism:	**Source Code:**	**Execution Space:**	**Location:**	**Side:**
Range Is Null Safe List Unsafe List Business Logic	Codebase Third Party Standard Library Processor	Userland Kernel Bare-Metal	User Entered Stored Transferred In Use	Client Server

Memory Addressing Bugs (MAD)



Causes

Improper Operation:
- Missing
- Mismatched
- Erroneous

Improper Pointer:
- NULL Pointer
- Wild Pointer
- Dangling Pointer
- Over Bounds
- Under Bounds
- Untrusted Pointer
- Wrong Position
- Hardcoded Address
- Casted Pointer

Improper Object:
- Wrong Size Used
- Not Enough Allocated

MAD Operations
- Initialize
- Reposition
- Reassign

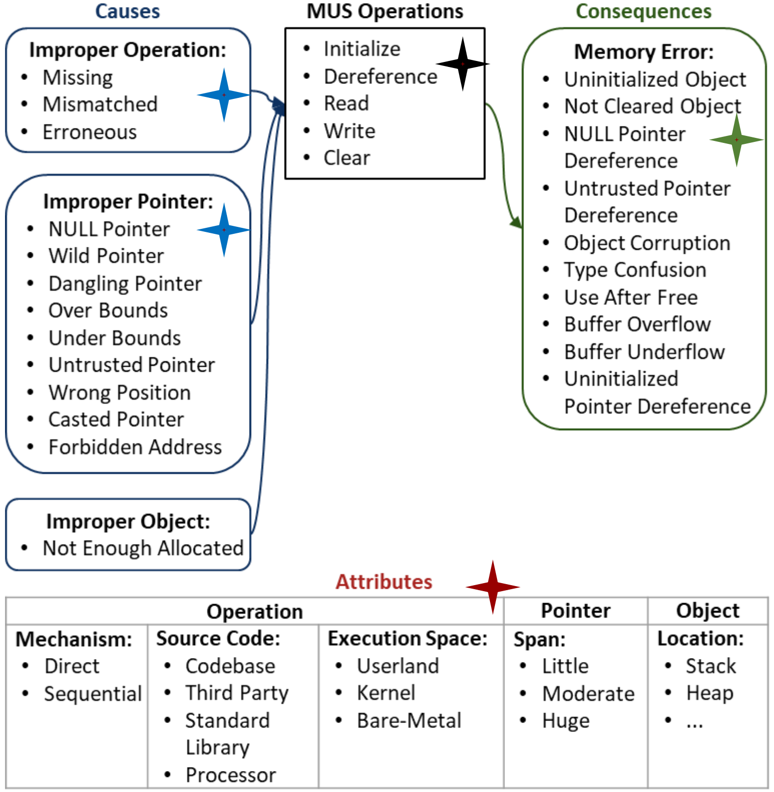
Consequences

Improper Pointer for Next Operation:
- NULL Pointer
- Wild Pointer
- Dangling Pointer
- Over Bounds
- Under Bounds
- Untrusted Pointer
- Wrong Position
- Casted Pointer
- Forbidden Address

Attributes

Operation			Object
Mechanism:	**Source Code:**	**Execution Space:**	**Location:**
Direct Sequential	Codebase Third Party Standard Library Processor	Userland Kernel Bare-Metal	Stack Heap ...

Memory Use Bugs (MUS)



Causes

Improper Operation:
- Missing
- Mismatched
- Erroneous

Improper Pointer:
- NULL Pointer
- Wild Pointer
- Dangling Pointer
- Over Bounds
- Under Bounds
- Untrusted Pointer
- Wrong Position
- Casted Pointer
- Forbidden Address

Improper Object:
- Not Enough Allocated

MUS Operations
- Initialize
- Dereference
- Read
- Write
- Clear

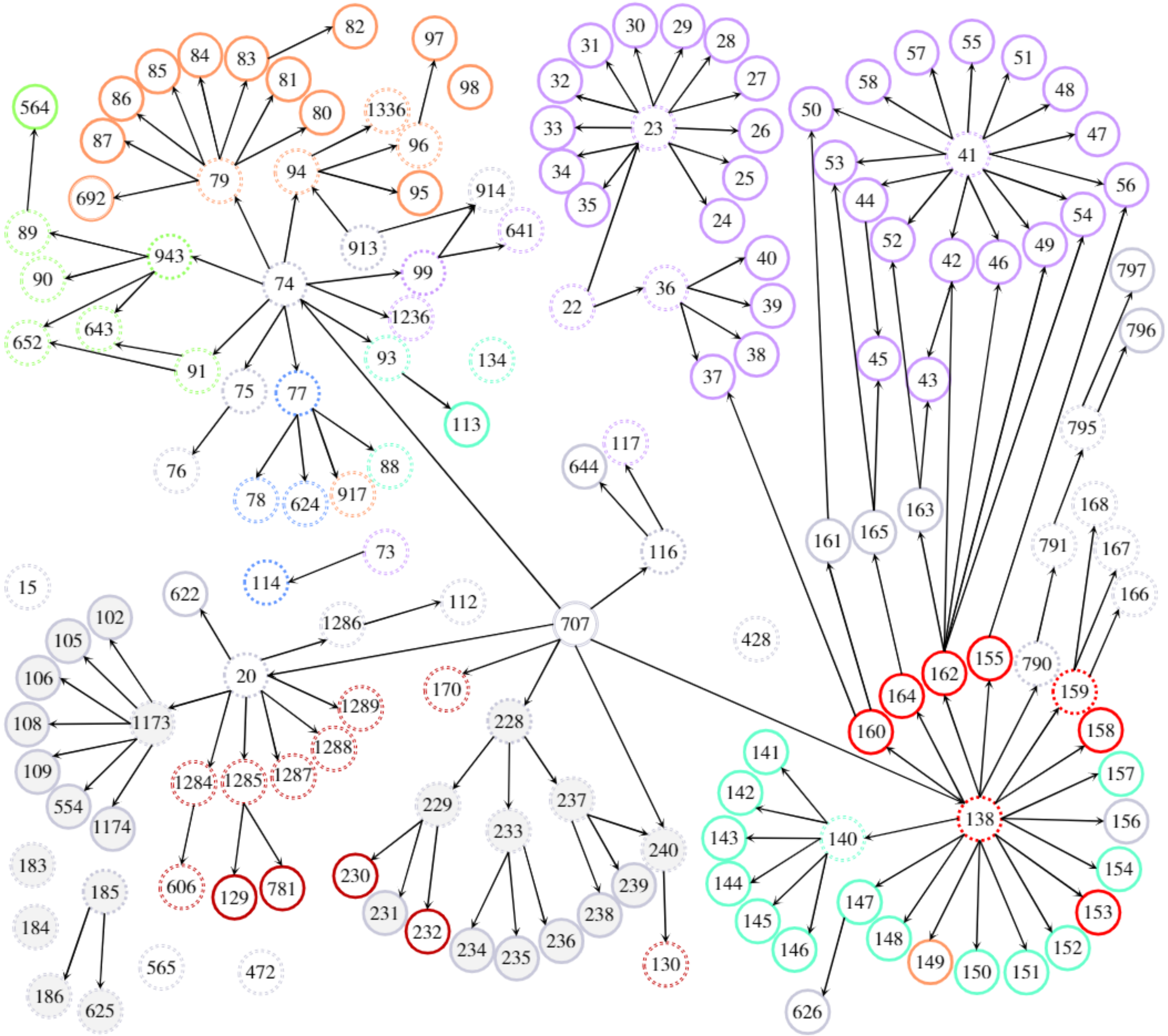
Consequences

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

Attributes

Operation			Pointer	Object
Mechanism:	**Source Code:**	**Execution Space:**	**Span:**	**Location:**
Direct Sequential	Codebase Third Party Standard Library Processor	Userland Kernel Bare-Metal	Little Moderate Huge	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 orDVR 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)

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

Alerts and Tips   Resources

ICS-CERT Advisories > Multiple RTOS (Update E)

ICS Advisory (ICSA-21-119-04)

Multiple RTOS (Update E)

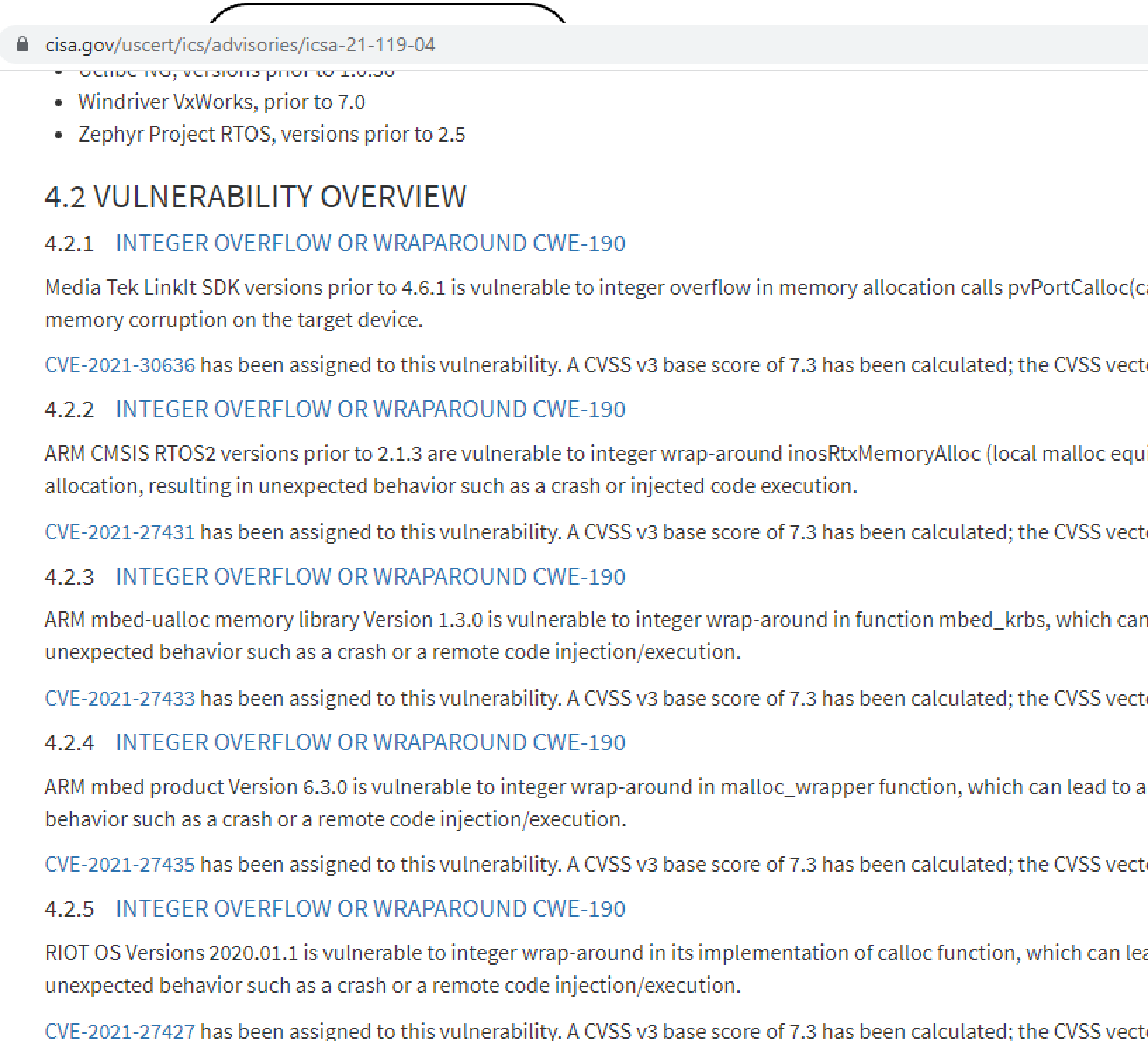
Original release date: April 19, 2022

Legal Notice

All information products included in https://us-cert.cisa.gov/ics are provided "as is" for informational purposes only. The Department of
regarding any information contained within. DHS does not endorse any commercial product or service, referenced in this product or othe
Light Protocol (TLP) marking in the header. For more information about TLP, see https://us-cert.cisa.gov/tlp/.

1. EXECUTIVE SUMMARY

- CVSS v3 9.8
- **ATTENTION:** Exploitable remotely/low attack complexity
- **Vendors:** Multiple



cisa.gov/uscert/ics/advisories/icsa-21-119-04

- Windriver VxWorks, prior to 7.0
- Zephyr Project RTOS, versions prior to 2.5

4.2 VULNERABILITY OVERVIEW

4.2.1 INTEGER OVERFLOW OR WRAPAROUND CWE-190

Media Tek LinkIt SDK versions prior to 4.6.1 is vulnerable to integer overflow in memory allocation calls pvPortCalloc(c
memory corruption on the target device.

CVE-2021-30636 has been assigned to this vulnerability. A CVSS v3 base score of 7.3 has been calculated; the CVSS vect

4.2.2 INTEGER OVERFLOW OR WRAPAROUND CWE-190

ARM CMSIS RTOS2 versions prior to 2.1.3 are vulnerable to integer wrap-around inosRtxMemoryAlloc (local malloc equ
allocation, 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 vect

4.2.3 INTEGER OVERFLOW OR WRAPAROUND CWE-190

ARM mbed-ualloc memory library Version 1.3.0 is vulnerable to integer wrap-around in function mbed_krbs, which can
unexpected behavior such as a crash or a 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 vect

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 a
behavior such as a crash or a 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 vect

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 lea
unexpected behavior such as a crash or a 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 vect

CVE-2021-21834 and the Bad Allocation Chain

Cause	DVR Operation	Consequence
Improper Operation: Missing	Check `(u64)ptr->nb_entries > (u64)SIZE_MAX/sizeof(u64))`	**Improper Data Value:** Inconsistent Value (`> max 64-bit int`)

Attributes

Mechanism:	Source Code:	Execution Space:	Data State:
- Range	- Third Party (library `box_code_base.c`)	- Local	- Stored (number of entries read from file)

Cause	TCM Operation	Consequence
Improper Data Value: Wrong Argument	Calculate `(ptr->nb_entries*sizeof(u64))`	**Improper Data Value:** Wrap Around

Attributes

Mechanism:	Source Code:	Data Value Kind:	Data Type Kind:
- Operator (Arithmetic: `'*'`)	- Third Party (library `box_code_base.c`)	- Numeric	- Structured

Cause	MAL Operation	Consequence
Improper Data Value: Wrong Size (size of memory to allocate)	Allocate (`gf_malloc()`)	**Improper Object Size:** Not Enough Allocated

Attributes

Mechanism:	Source Code:	Execution Space:	Pointer Ownership:	Object Location:
- Explicit	- Third Party (library `box_code_base.c`)	- Userland	- Single	- Heap

Cause	MAD Operation	Consequence
Improper Object Size: Not Enough Allocated	Reposition	**Improper Data Value:** Over Bounds Pointer

Attributes

Mechanism:	Source Code:	Execution Space:	Object Location:
- Sequential	- Third Party (library `box_code_base.c`)	- Userland	- Heap

Cause	MUS Operation	Consequence
Improper Data Value: Over Bounds Pointer	Write	**Memory Error:** Buffer Overflow

Attributes

Mechanism:	Source Code:	Execution Space:	Pointer Span:	Object Location:
- Sequential	- Third Party (library `box_code_base.c`)	- Userland	- Huge	- Heap

BF Tools Set

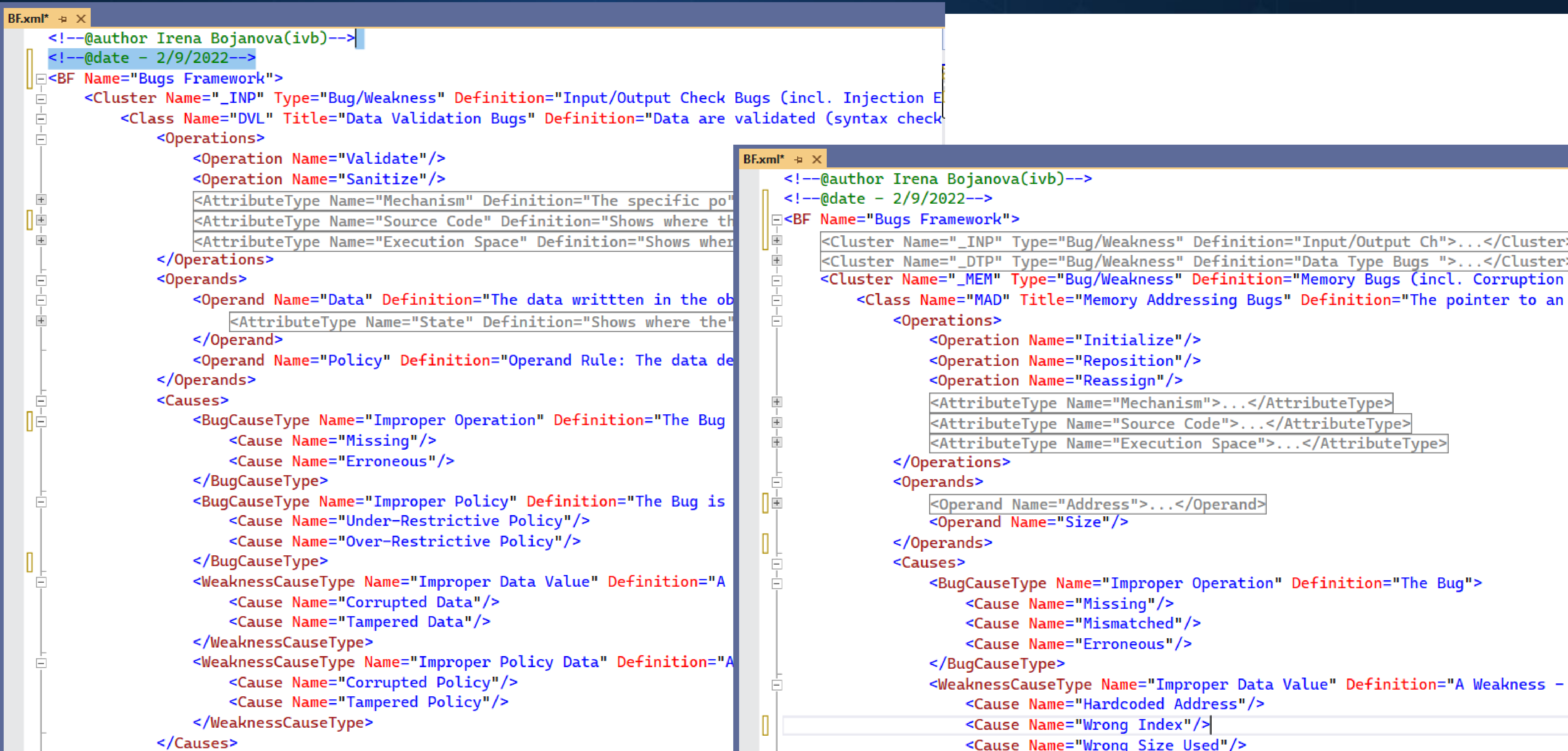
I. Editor of BF Vulnerabilities Descriptions

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.

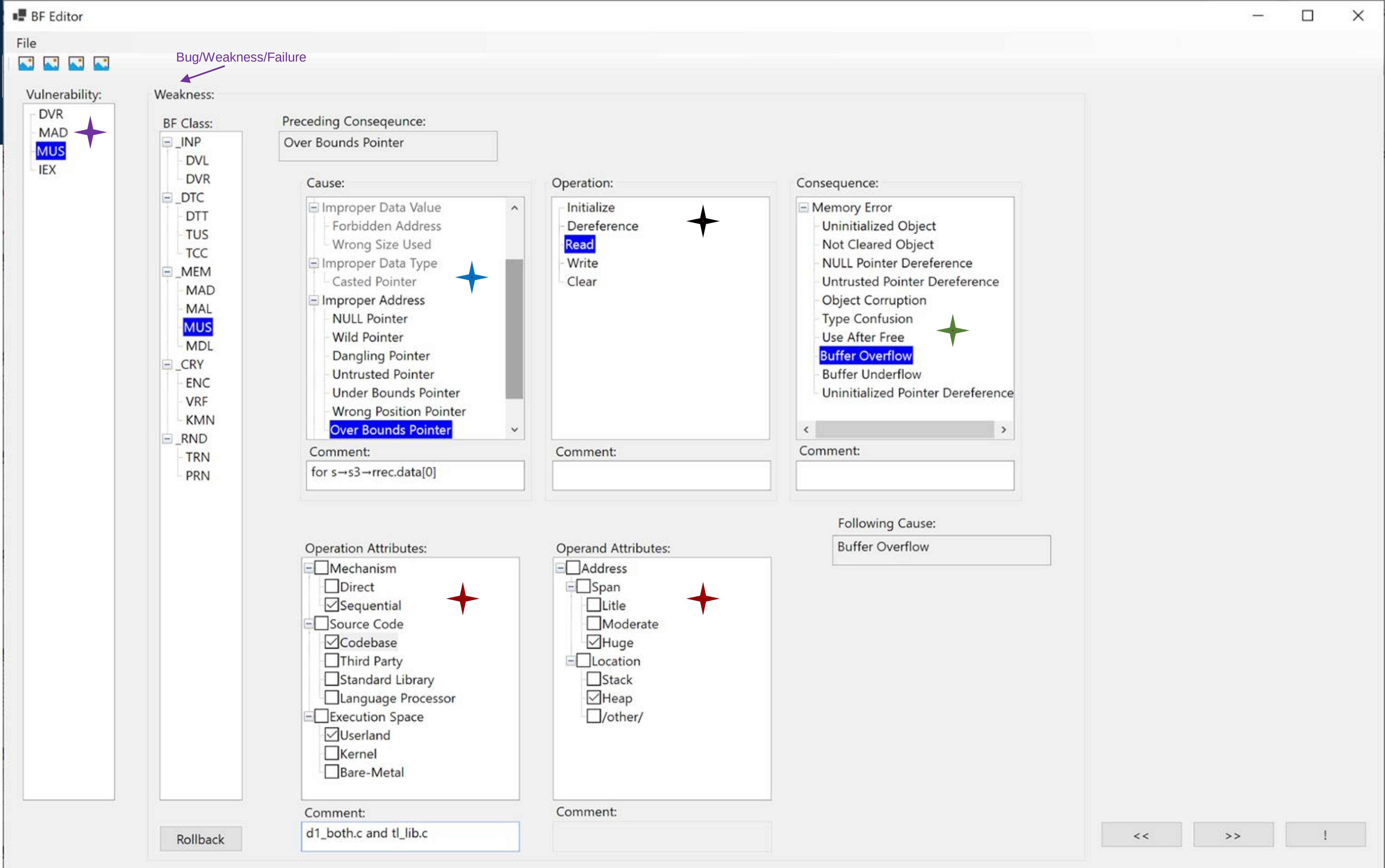
1. BF.xml – all BF Clusters of Classes



```
<!--@author Irena Bojanova(ivb)-->
<!--@date - 2/9/2022-->
<BF Name="Bugs Framework">
    <Cluster Name="_INP" Type="Bug/Weakness" Definition="Input/Output Check Bugs (incl. Injection E
        <Class Name="DVL" Title="Data Validation Bugs" Definition="Data are validated (syntax check
            <Operations>
                <Operation Name="Validate"/>
                <Operation Name="Sanitize"/>
                <AttributeType Name="Mechanism" Definition="The specific po"
                <AttributeType Name="Source Code" Definition="Shows where th
                <AttributeType Name="Execution Space" Definition="Shows wher
            </Operations>
            <Operands>
                <Operand Name="Data" Definition="The data writtten in the ob
                    <AttributeType Name="State" Definition="Shows where the"
                </Operand>
                <Operand Name="Policy" Definition="Operand Rule: The data de
            </Operands>
            <Causes>
                <BugCauseType Name="Improper Operation" Definition="The Bug
                    <Cause Name="Missing"/>
                    <Cause Name="Erroneous"/>
                </BugCauseType>
                <BugCauseType Name="Improper Policy" Definition="The Bug is
                    <Cause Name="Under-Restrictive Policy"/>
                    <Cause Name="Over-Restrictive Policy"/>
                </BugCauseType>
                <WeaknessCauseType Name="Improper Data Value" Definition="A
                    <Cause Name="Corrupted Data"/>
                    <Cause Name="Tampered Data"/>
                </WeaknessCauseType>
                <WeaknessCauseType Name="Improper Policy Data" Definition="A
                    <Cause Name="Corrupted Policy"/>
                    <Cause Name="Tampered Policy"/>
                </WeaknessCauseType>
            </Causes>
```

```
<!--@author Irena Bojanova(ivb)-->
<!--@date - 2/9/2022-->
<BF Name="Bugs Framework">
    <Cluster Name="_INP" Type="Bug/Weakness" Definition="Input/Output Ch">...</Cluster
    <Cluster Name="_DTP" Type="Bug/Weakness" Definition="Data Type Bugs ">...</Cluster
    <Cluster Name="_MEM" Type="Bug/Weakness" Definition="Memory Bugs (incl. Corruption
        <Class Name="MAD" Title="Memory Addressing Bugs" Definition="The pointer to an
            <Operations>
                <Operation Name="Initialize"/>
                <Operation Name="Reposition"/>
                <Operation Name="Reassign"/>
                <AttributeType Name="Mechanism">...</AttributeType>
                <AttributeType Name="Source Code">...</AttributeType>
                <AttributeType Name="Execution Space">...</AttributeType>
            </Operations>
            <Operands>
                <Operand Name="Address">...</Operand>
                <Operand Name="Size"/>
            </Operands>
            <Causes>
                <BugCauseType Name="Improper Operation" Definition="The Bug">
                    <Cause Name="Missing"/>
                    <Cause Name="Mismatched"/>
                    <Cause Name="Erroneous"/>
                </BugCauseType>
                <WeaknessCauseType Name="Improper Data Value" Definition="A Weakness -
                    <Cause Name="Hardcoded Address"/>
                    <Cause Name="Wrong Index"/>
                    <Cause Name="Wrong Size Used"/>
```

1.



Vulnerability:
- DVR
- MAD
- **MUS**
- IEX

Weakness:

BF Class:
- _INP
  - DVL
  - DVR
- _DTC
  - DTT
  - TUS
  - TCC
- _MEM
  - MAD
  - MAL
  - **MUS**
  - MDL
- _CRY
  - ENC
  - VRF
  - KMN
- _RND
  - TRN
  - PRN

Preceding Conseqeunce: Over Bounds Pointer

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

Comment: for s→s3→rrec.data[0]

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

Comment:

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

Comment:

Following Cause: Buffer Overflow

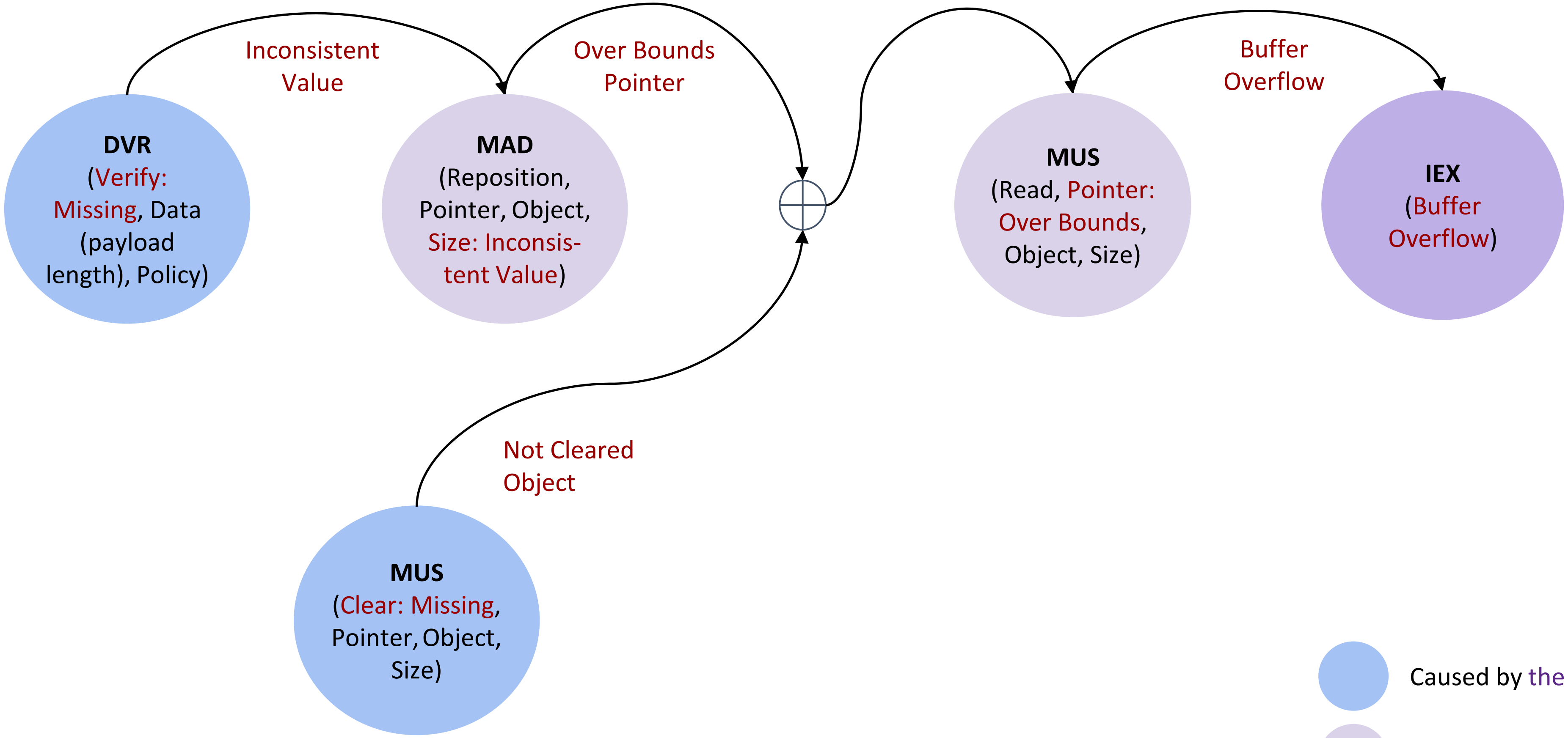
Operation Attributes:
- ☐ Mechanism
  - ☐ Direct
  - ☑ Sequential
- ☐ Source Code
  - ☑ Codebase
  - ☐ Third Party
  - ☐ Standard Library
  - ☐ Language Processor
- ☐ Execution Space
  - ☑ Userland
  - ☐ Kernel
  - ☐ Bare-Metal

Comment: d1_both.c and tl_lib.c

Operand Attributes:
- ☐ Address
  - ☐ Span
    - ☐ Litle
    - ☐ Moderate
    - ☑ Huge
  - ☐ Location
    - ☐ Stack
    - ☑ Heap
    - ☐ /other/

Comment:

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
(Clear: Missing, Pointer, Object, Size)

Not Cleared Object

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

Buffer Overflow

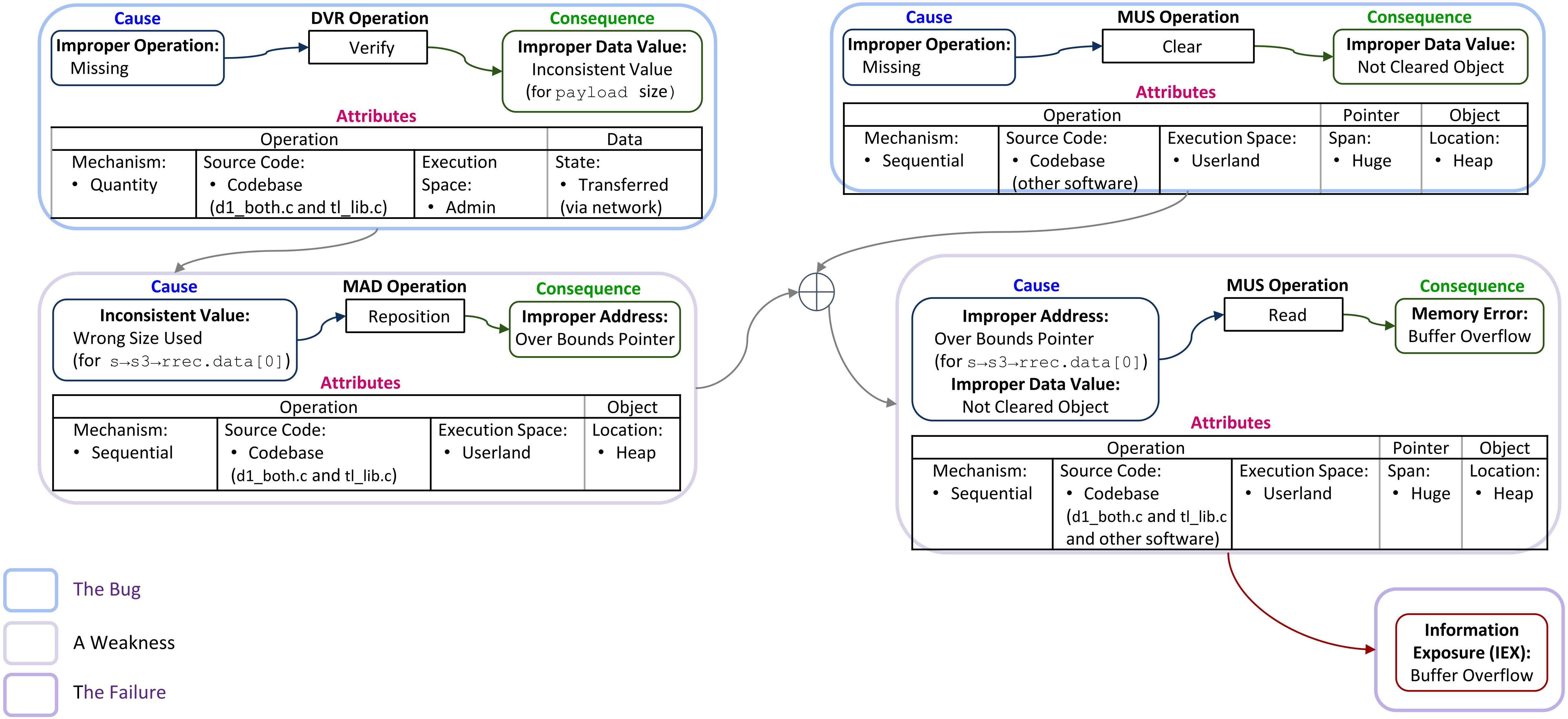
IEX
(Buffer Overflow)

● 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

DVR Operation

Verify

Consequence

Improper Data Value: Inconsistent Value (for `payload` size)

Attributes

Operation			Data
Mechanism: • Quantity	Source Code: • Codebase (d1_both.c and tl_lib.c)	Execution Space: • Admin	State: • Transferred (via network)

Cause

Improper Operation: Missing

MUS Operation

Clear

Consequence

Improper Data Value: Not Cleared Object

Attributes

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

Cause

Inconsistent Value: Wrong Size Used (for `s→s3→rrec.data[0]`)

MAD Operation

Reposition

Consequence

Improper Address: Over Bounds Pointer

Attributes

Operation			Object
Mechanism: • Sequential	Source Code: • Codebase (d1_both.c and tl_lib.c)	Execution Space: • Userland	Location: • Heap

Cause

Improper Address: Over Bounds Pointer (for `s→s3→rrec.data[0]`)

Improper Data Value: Not Cleared Object

MUS Operation

Read

Consequence

Memory Error: Buffer Overflow

Attributes

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

Information Exposure (IEX): Buffer Overflow

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.

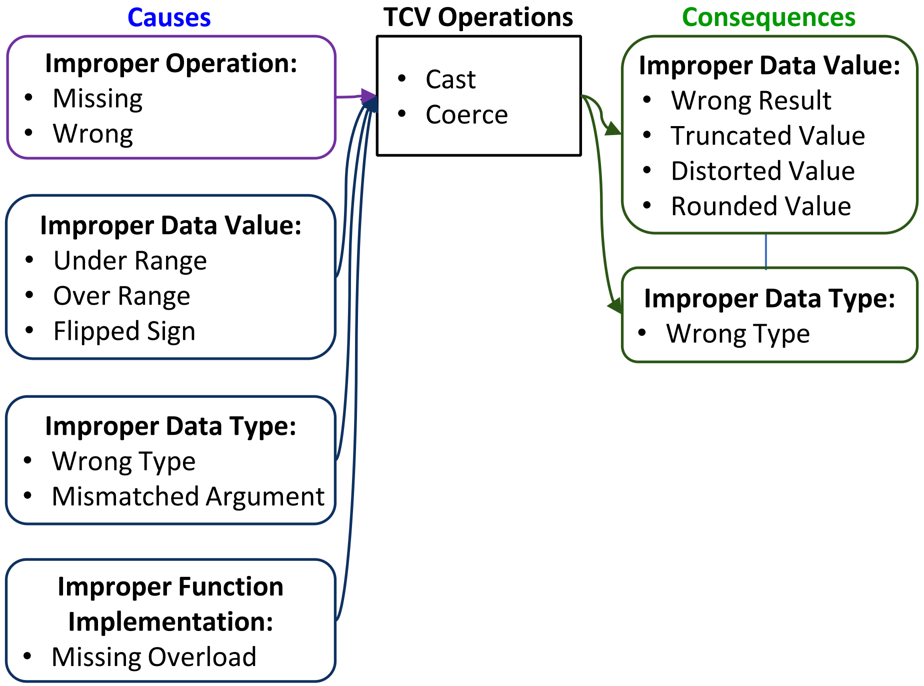
1. BF Classes and Matrices of Cause-Operation-Consequences

- 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
- Mismatched Argument

Improper Function Implementation:
- Missing Overload

TCV Operations
- Cast
- Coerce

Consequences

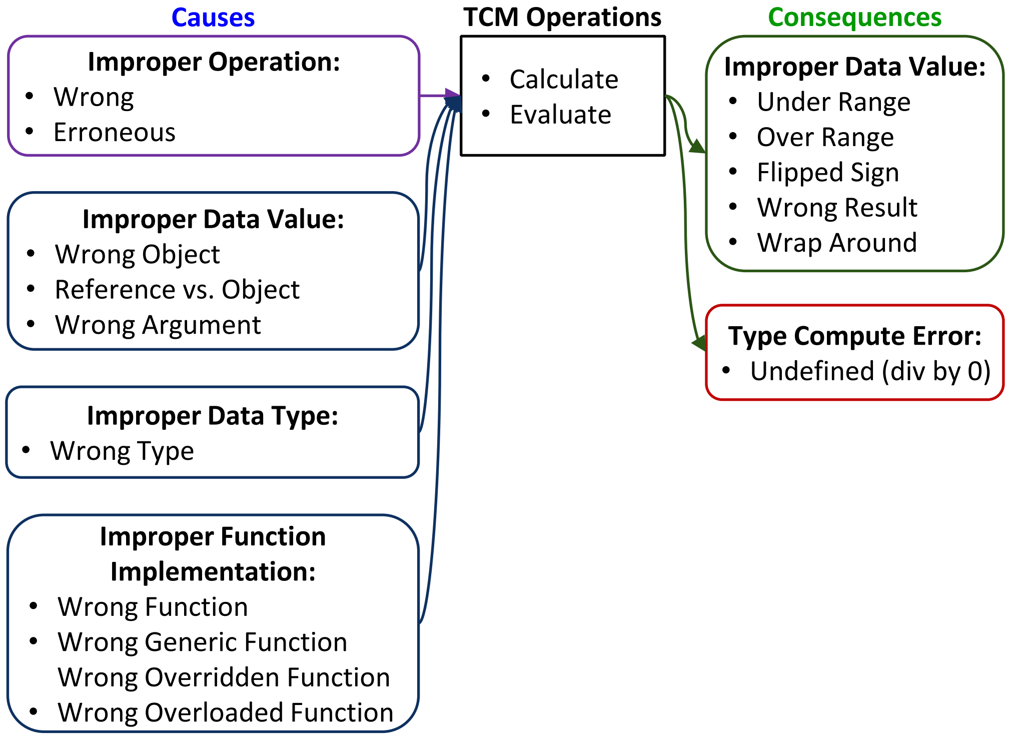
Improper Data Value:
- Wrong Result
- Truncated Value
- Distorted Value
- Rounded Value

Improper Data Type:
- Wrong Type

Attributes

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

Type Compute Bugs (TCM)



Causes

Improper Operation:
- Wrong
- Erroneous

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

TCM Operations
- Calculate
- Evaluate

Consequences

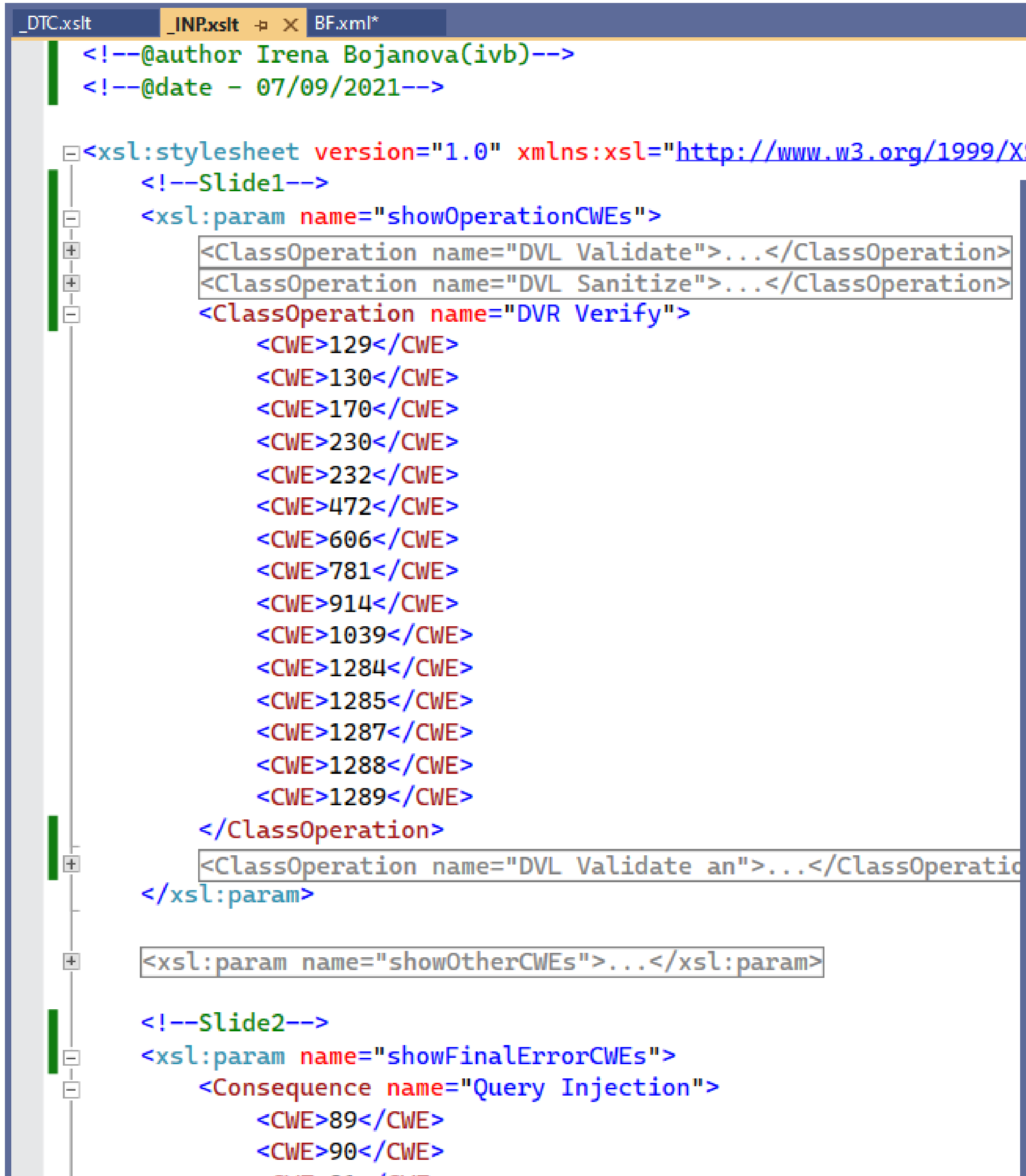
Improper Data Value:
- Under Range
- Over Range
- Flipped Sign
- Wrong Result
- Wrap Around

Type Compute Error:
- Undefined (div by 0)

Attributes

Mechanism:	Source Code:	Data Value Kind:	Data Type Kind:
• Function • Operator • Method • Lambda Expression • Procedure	• Codebase • Third Party • Standard Library • Processor	• Numeric • Text • Pointer • Boolean	• Primitive • Structured

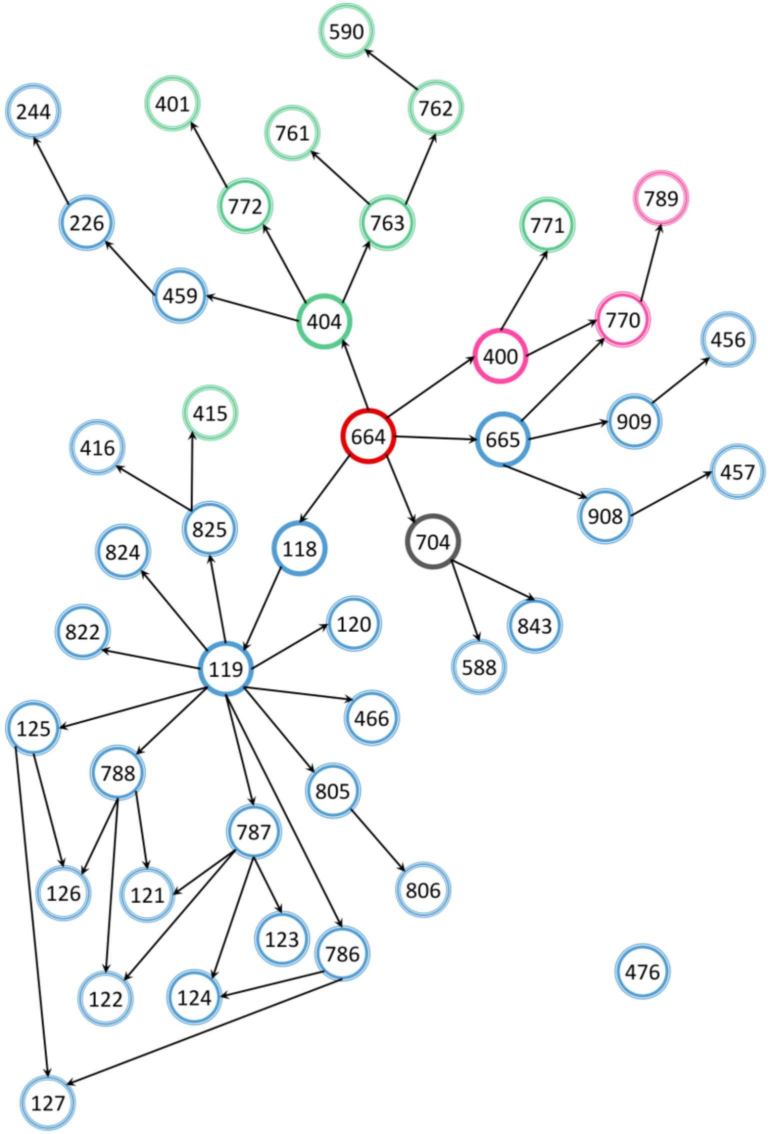
3. CWEs Relate to BF Clusters

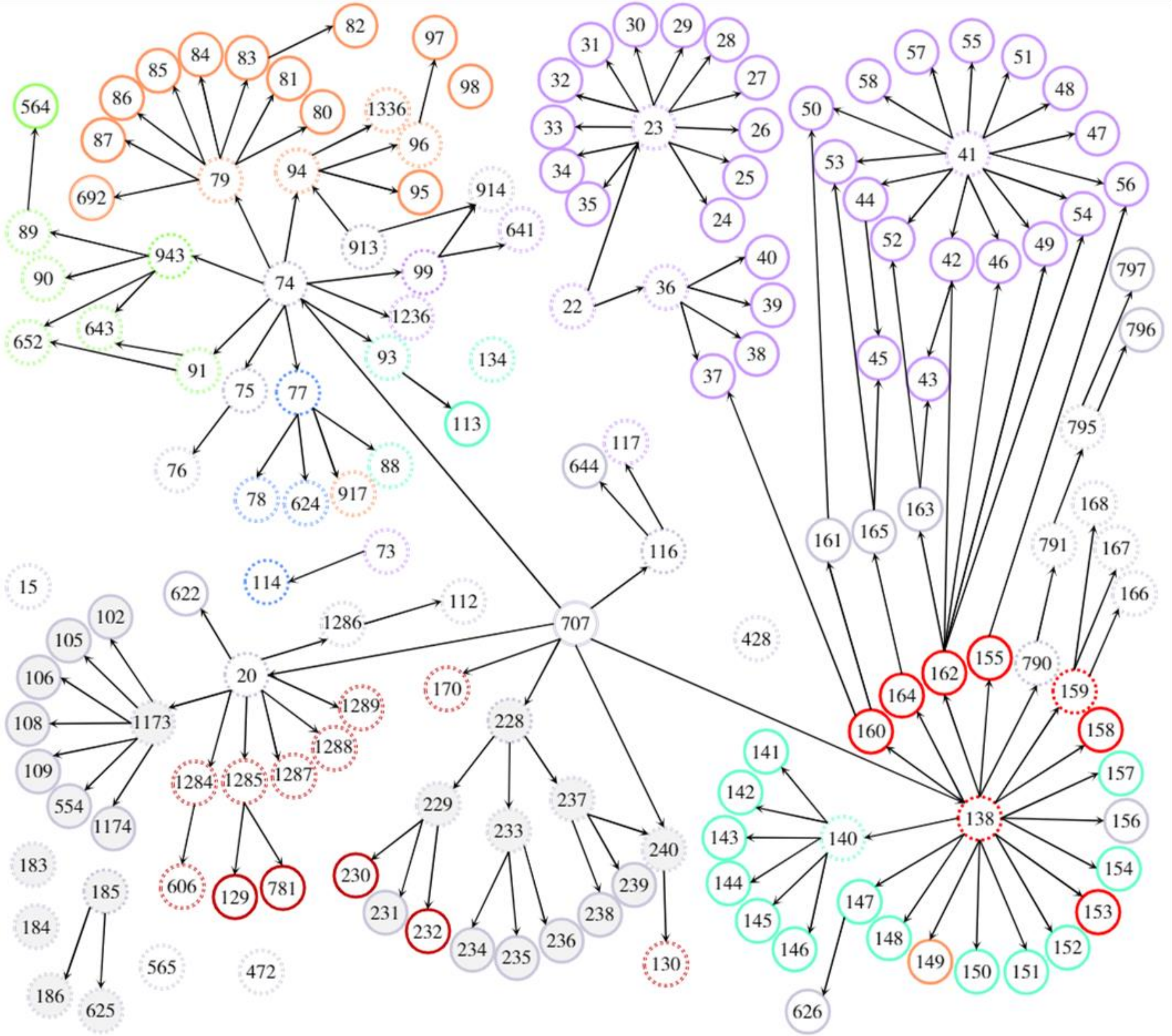


```
<!--@author Irena Bojanova(ivb)-->
<!--@date - 07/09/2021-->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:msxsl="urn:sch
    <!--Slide1-->
    <xsl:param name="showOperationCWEs">
        <ClassOperation name="DVL Validate">...</ClassOperation>
        <ClassOperation name="DVL Sanitize">...</ClassOperation>
        <ClassOperation name="DVR Verify">
            <CWE>129</CWE>
            <CWE>130</CWE>
            <CWE>170</CWE>
            <CWE>230</CWE>
            <CWE>232</CWE>
            <CWE>472</CWE>
            <CWE>606</CWE>
            <CWE>781</CWE>
            <CWE>914</CWE>
            <CWE>1039</CWE>
            <CWE>1284</CWE>
            <CWE>1285</CWE>
            <CWE>1287</CWE>
            <CWE>1288</CWE>
            <CWE>1289</CWE>
        </ClassOperation>
        <ClassOperation name="DVL Validate an">...</ClassOperatic
    </xsl:param>

    <xsl:param name="showOtherCWEs">...</xsl:param>

    <!--Slide2-->
    <xsl:param name="showFinalErrorCWEs">
        <Consequence name="Query Injection">
            <CWE>89</CWE>
            <CWE>90</CWE>
```

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:msxsl="urn::
<!-- could list repeating CWEs in all sets and color with mix of the colors-->
    <!--Slide1-->
    <xsl:param name="showOperationCWEs">
        <ClassOperation name="General">...</ClassOperation>

        <ClassOperation name="DCL Declare">
            <CWE>471</CWE>
            <CWE>491</CWE>
            <CWE>492</CWE>
            <CWE>493</CWE>
            <CWE>495</CWE>
            <CWE>496</CWE>
            <CWE>499</CWE>
            <CWE>500</CWE>
            <CWE>582</CWE>
            <CWE>583</CWE>
            <CWE>608</CWE>
            <CWE>766</CWE>
        </ClassOperation>
        <ClassOperation name="DCL Define">...</ClassOperation>
        <ClassOperation name="NRS Refer">
            <!--<CWE>386</CWE>-->
            <CWE>706</CWE>
        </ClassOperation>
        <ClassOperation name="NRS Call">
            <CWE>386</CWE>
        </ClassOperation>
        <ClassOperation name="TCV Cast">
            <CWE>588</CWE>
```

Generated Graphical Representations of the Input/Output Cljuster Mappings to CWE



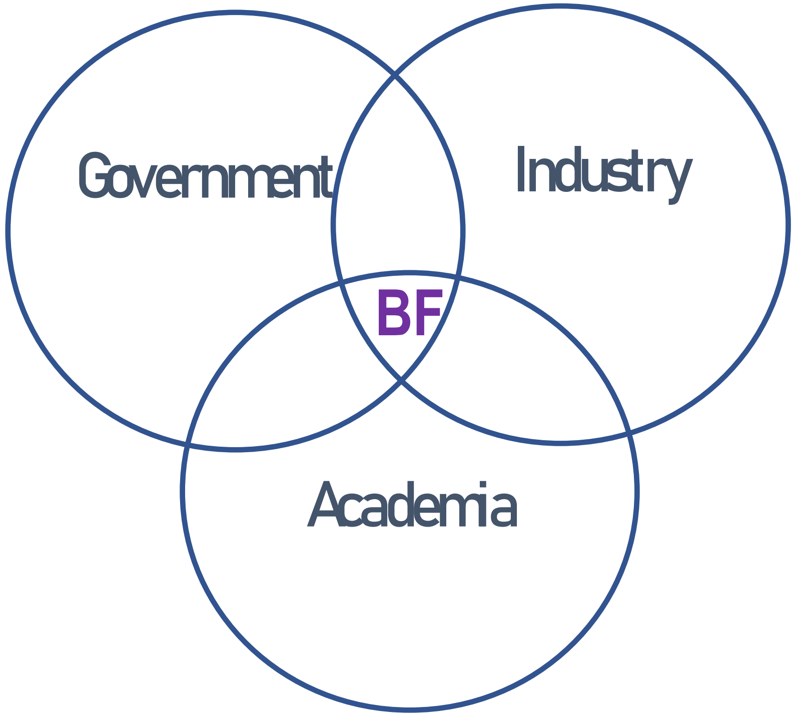


BF – Potential Impact

BF – Potential Impacts

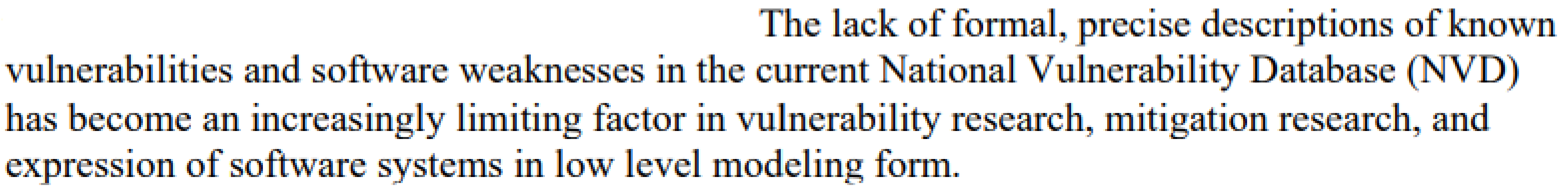
- Allow precise communication about software bugs and weaknesses

- Help identify exploit mitigation techniques

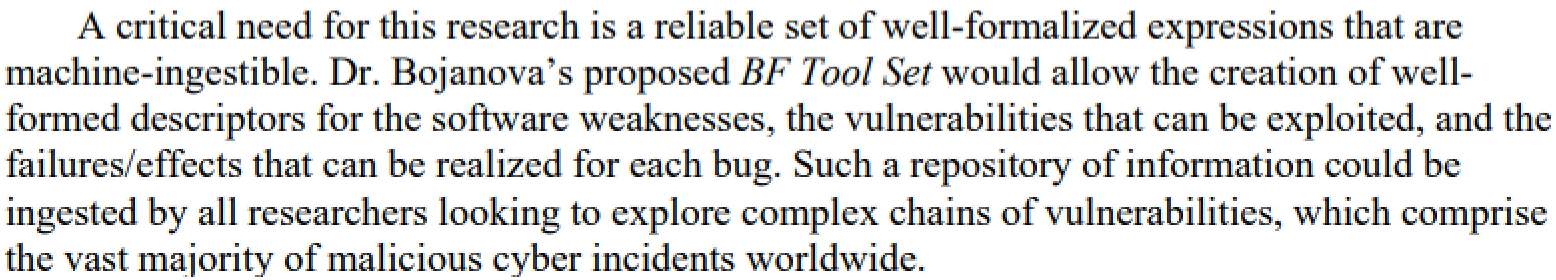


BF Addresses a Unique Need

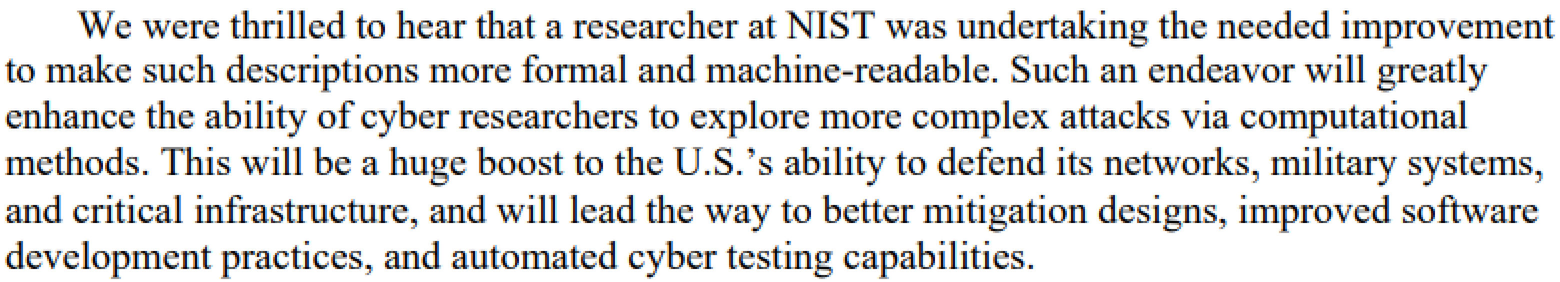
- JHU APL – Automated Vulnerability Testing via Executable Attack Graphs:
  - Chain vulnerabilities via logical directed graphs
  - Determine most mitigation “paths” with least changes
  - Detect user behavior prior to malicious effect



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.



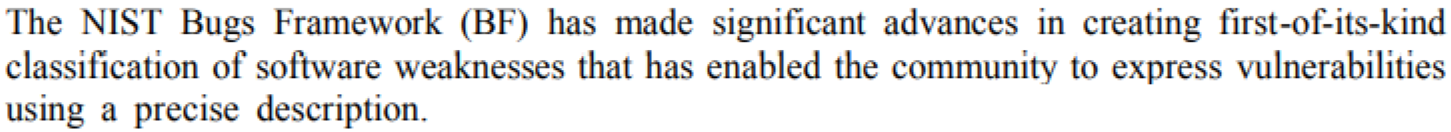
A critical need for this research is a reliable set of well-formalized expressions that are machine-ingestible. Dr. Bojanova’s proposed *BF Tool Set* would allow the creation of well-formed descriptors for the software weaknesses, the vulnerabilities that can be exploited, and the failures/effects that 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.



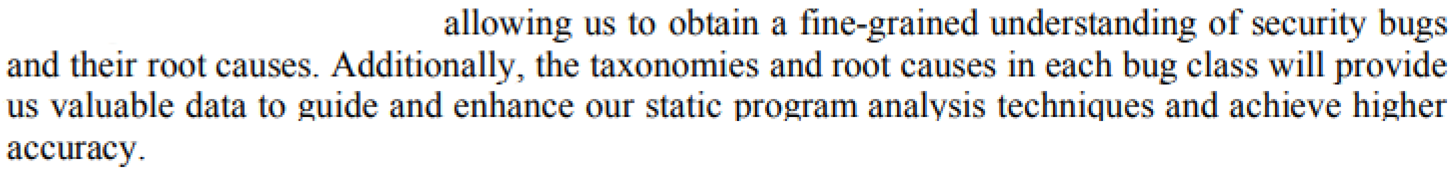
We were thrilled to hear that a researcher at NIST was undertaking the needed improvement to make such descriptions more formal and machine-readable. Such an endeavor will greatly enhance the ability of cyber researchers to explore more complex attacks via computational methods. This will be a huge boost 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

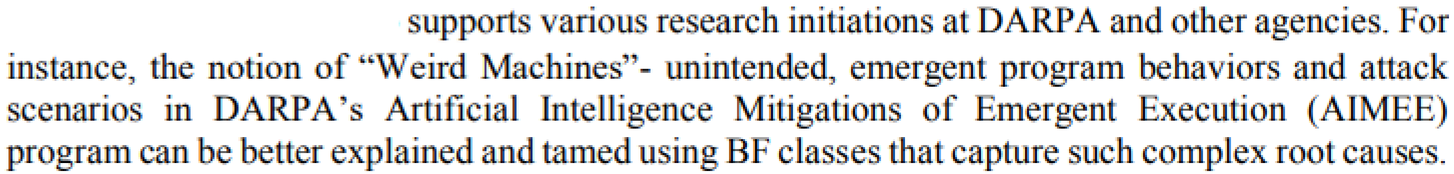
- RIT [Secure and Trustworthy Cyberspace (SaTC)]:
  - Projects on Vulnerabilities Research



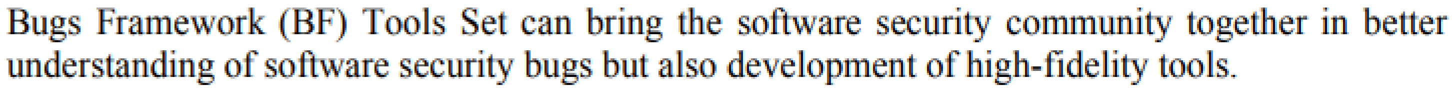
The NIST Bugs Framework (BF) has made significant advances in creating first-of-its-kind classification of software weaknesses that has enabled the community to express vulnerabilities using a precise description.



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.

Latest BF Publications

2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)

Classifying Memory Bugs Using Bugs Framework Approach

Irena Bojanova
SSD, ITL
NIST
Gaithersburg, MD, USA
irena.bojanova@nist.gov

Carlos Eduardo Galhardo
Dimel, Sinst
INMETRO
Duque de Caxias, RJ, Brazil
cegalhardo@inmetro.gov.br

***Abstract*—In this work, we present an orthogonal classification of memory corruption bugs, allowing precise structured descriptions of related software vulnerabilities. The Common Weakness Enumeration (CWE) is a well-known and used list of software weaknesses. However, it's exhaustive list approach is prone to gaps and overlaps in coverage. Instead, we utilize the Bugs Framework (BF) approach to define language-independent classes that cover all possible kinds of memory corruption bugs. Each class is a taxonomic category of a weakness type, defined by sets of operations, cause→consequence relations, and attributes. A BF description of a bug or a weakness is an instance of a taxonomic BF class, with one operation, one cause, one consequence, and their attributes. Any memory vulnerability then can be described as a chain of such instances and their consequence–cause transitions. We showcase that BF is a classification system that extends the CWE, providing a structured way to precisely describe real world vulnerabilities. It allows clear communication about software bugs and weaknesses and can help identify exploit mitigation techniques.**

***Keywords*—Bug classification, bug taxonomy, software vulnerability, software weakness, memory corruption.**

I. INTRODUCTION

Software bugs in memory allocation, use, and deallocation may lead to memory corruption and memory disclosure, opening doors for cyberattacks. Classifying them would allow precise communication and help us teach about them, understand and identify them, and avoid security failures. For that, we utilize the Bug Framework (BF) approach [1].

The Common Weakness Enumeration (CWE) [2] and the Common Vulnerabilities and Exposures (CVE) [3] are well-known and used lists of software security weaknesses and vulnerabilities. However, the CWE's exhaustive list approach is prone to having gaps and overlaps in coverage, as demonstrated by the National Vulnerability Database (NVD) effort to link CVEs to appropriate CWEs [4]. Instead, we utilize the BF approach to define four language-independent, orthogonal classes that cover all possible kinds of memory related software bugs and weaknesses: Memory Allocation Bugs (MAL), Memory Use Bugs (MUS), Memory Deallocation Bugs (MDL), and Memory Addressing Bugs (MAD). This BF Memory Bugs taxonomy can be viewed as a structured extension to the memory-related CWEs, allowing bug reporting tools to produce more detailed, precise, and unambiguous descriptions of identified memory bugs.

In this paper, we first summarize the latest BF approach and methodology. Next, we analyze the types of memory corruption bugs and define the BF Memory Bugs Model. Then, we present our BF memory bugs classes and showcase they provide a better, structured way to describe CVE entries [3]. We identify the corresponding clusters of memory corruption CWEs and their relations to the BF classes. Finally, we discuss the use of these new BF classes for identifying exploit mitigation techniques.

II. BF APPROACH AND METHODOLOGY

BF's approach is different from CWE's exhaustive list approach. BF is a classification. Each BF class is a taxonomic category of a weakness type. It relates to a distinct phase of software execution, the operations specific for that phase and the operands required as input to those operations.

We define a software bug as a coding error that needs to be fixed. A weakness is caused by a bug or ill-formed data. A weakness type is also a meaningful notion, as different vulnerabilities may have the same type of underlying weaknesses. We define a vulnerability as an instance of a weakness type that leads to a security failure. It may have more than one underlying weaknesses linked by causality.

BF describes a bug or a weakness as an improper state and its transition. The transition is to another weakness or to a failure. An improper state is defined by the tuple $(\mathtt{operation}, \mathtt{operand}_1, \cdots, \mathtt{operand}_n)$, where at least one element is improper. The initial state is always caused by a bug; a coding error within the operation, which if fixed will resolve the vulnerability. An intermediate state is caused by ill-formed data; it has at least one improper operand. Rarely an intermediate state may also have a bug, which if fixed will also resolve the vulnerability. The final state, the failure, is caused by a final error (undefined or exploitable system behavior), which usually directly relates to a CWE [2]. A transition is the result of the operation over the operands.

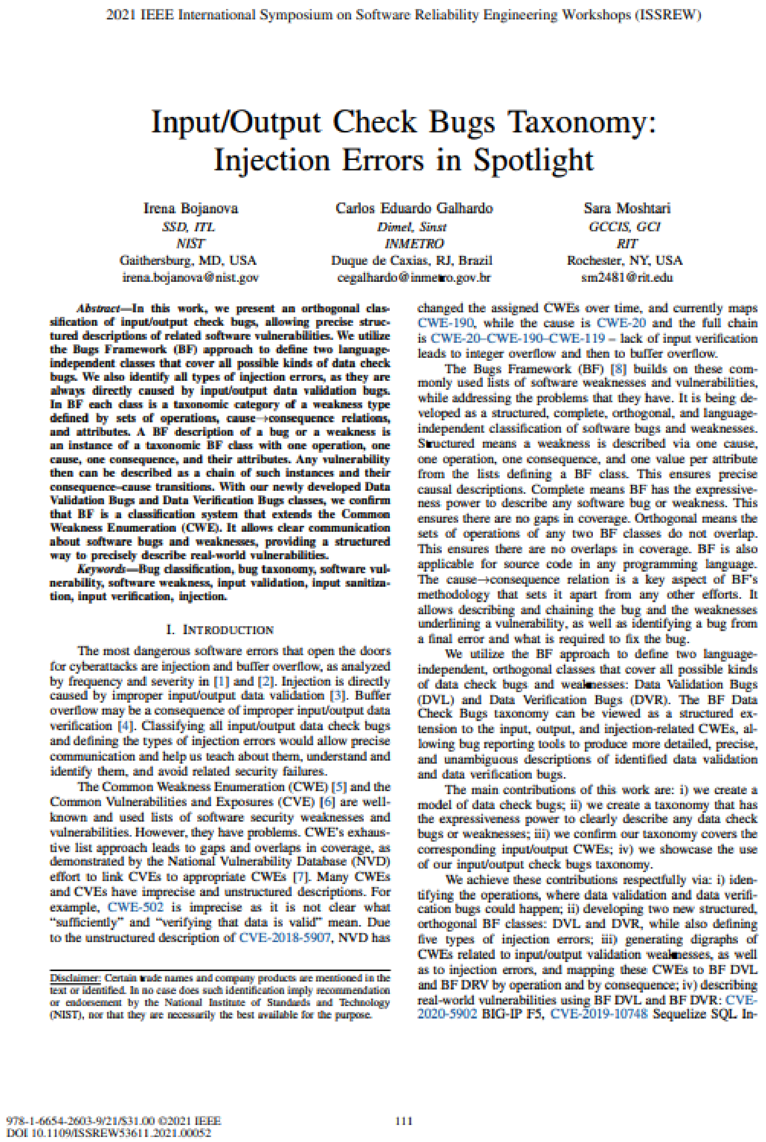
BF describes a vulnerability as a chain of improper states and their transitions. Each improper state is an instance of a BF class. The transition from the initial state is by improper operation over proper operands. The transitions from intermediate states are by proper operations with at least one improper operand.

In some cases, several vulnerabilities have to be present for an exploit to be harmful. The final errors resulting from different chains converge to cause a failure. The bug in at least one of the chains must be fixed to avoid that failure.

We call a BF class the set of operations, the valid cause→consequence relations for these operations, their at-

---

978-1-6654-2463-9/21/$31.00 ©2021 IEEE
DOI 10.1109/COMPSAC51774.2021.00159



2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)

Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight

Irena Bojanova
SSD, ITL
NIST
Gaithersburg, MD, USA
irena.bojanova@nist.gov

Carlos Eduardo Galhardo
Dimel, Sinst
INMETRO
Duque de Caxias, RJ, Brazil
cegalhardo@inmetro.gov.br

Sara Moshtari
GCCIS, GCI
RIT
Rochester, NY, USA
sm2481@rit.edu

***Abstract*—In this work, we present an orthogonal classification of input/output check bugs, allowing precise structured descriptions of related software vulnerabilities. We utilize the Bugs Framework (BF) approach to define two language-independent classes that cover all possible kinds of data check bugs. We also identify all types of injection errors, as they are always directly caused by input/output data validation bugs. In BF each class is a taxonomic category of a weakness type defined by sets of operations, cause→consequence relations, and attributes. A BF description of a bug or a weakness is an instance of a taxonomic BF class with one operation, one cause, one consequence, and their attributes. Any vulnerability then can be described as a chain of such instances and their consequence–cause transitions. With our newly developed Data Validation Bugs and Data Verification Bugs classes, we confirm that BF is a classification system that extends the Common Weakness Enumeration (CWE). It allows clear communication about software bugs and weaknesses, providing a structured way to precisely describe real-world vulnerabilities.**

***Keywords*—Bug classification, bug taxonomy, software vulnerability, software weakness, input validation, input sanitization, input verification, injection.**

I. INTRODUCTION

The most dangerous software errors that open the doors for cyberattacks are injection and buffer overflow, as analyzed by frequency and severity in [1] and [2]. Injection is directly caused by improper input/output data validation [3]. Buffer overflow may be a consequence of improper input/output data verification [4]. Classifying all input/output data check bugs and defining the types of injection errors would allow precise communication and help us teach about them, understand and identify them, and avoid related security failures.

The Common Weakness Enumeration (CWE) [5] and the Common Vulnerabilities and Exposures (CVE) [6] are well-known and used lists of software security weaknesses and vulnerabilities. However, they have problems. CWE's exhaustive list approach leads to gaps and overlaps in coverage, as demonstrated by the National Vulnerability Database (NVD) effort to link CVEs to appropriate CWEs [7]. Many CWEs and CVEs have imprecise and unstructured descriptions. For example, CWE-502 is imprecise as it is not clear what "sufficiently" and "verifying that data is valid" mean. Due to the unstructured description of CVE-2018-5907, NVD has changed the assigned CWEs over time, and currently maps CWE-190, while the cause is CWE-20 and the full chain is CWE-20–CWE-190–CWE-119 – lack of input verification leads to integer overflow and then to buffer overflow.

The Bugs Framework (BF) [8] builds on these commonly used lists of software weaknesses and vulnerabilities, while addressing the problems that they have. It is being developed as a structured, complete, orthogonal, and language-independent classification of software bugs and weaknesses. Structured means a weakness is described via one cause, one operation, one consequence, and one value per attribute from the lists defining a BF class. This ensures precise causal descriptions. Complete means BF has the expressiveness power to describe any software bug or weakness. This ensures there are no gaps in coverage. Orthogonal means the sets of operations of any two BF classes do not overlap. This ensures there are no overlaps in coverage. BF is also applicable for source code in any programming language. The cause→consequence relation is a key aspect of BF's methodology that sets it apart from any other efforts. It allows describing and chaining the bug and the weaknesses underlining a vulnerability, as well as identifying a bug from a final error and what is required to fix the bug.

We utilize the BF approach to define two language-independent, orthogonal classes that cover all possible kinds of data check bugs and weaknesses: Data Validation Bugs (DVL) and Data Verification Bugs (DVR). The BF Data Check Bugs taxonomy can be viewed as a structured extension to the input, output, and injection-related CWEs, allowing bug reporting tools to produce more detailed, precise, and unambiguous descriptions of identified data validation and data verification bugs.

The main contributions of this work are: i) we create a model of data check bugs; ii) we create a taxonomy that has the expressiveness power to clearly describe any data check bugs or weaknesses; iii) we confirm our taxonomy covers the corresponding input/output CWEs; iv) we showcase the use of our input/output check bugs taxonomy.

We achieve these contributions respectfully via: i) identifying the operations, where data validation and data verification bugs could happen; ii) developing two new structured, orthogonal BF classes: DVL and DVR, while also defining five types of injection errors; iii) generating digraphs of CWEs related to input/output validation weaknesses, as well as to injection errors, and mapping these CWEs to BF DVL and BF DRV by operation and by consequence; iv) describing real-world vulnerabilities using BF DVL and BF DVR: CVE-2020-5902 BIG-IP F5, CVE-2019-10748 Sequelize SQL In-

---

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00052

Questions

Irena Bojanova: irena.bojanova@nist.gov