NIST Special Publication 800
NIST SP 800-231

# Bugs Framework (BF)

*Formalizing Cybersecurity Weaknesses and Vulnerabilities*

Irena Bojanova

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-231

NATIONAL INSTITUTE OF
STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

**NIST Special Publication 800**
**NIST SP 800-231**

# Bugs Framework (BF)

*Formalizing Cybersecurity Weaknesses and Vulnerabilities*

Irena Bojanova
*Software and Systems Division*
*Information Technology Laboratory*

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-231

July 2024

U.S. Department of Commerce
*Gina M. Raimondo, Secretary*

National Institute of Standards and Technology
*Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology*

**Authority**

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

**NIST Technical Series Policies**

**Publication History**

Approved by the NIST Editorial Review Board on 2024-05-16

**How to cite this NIST Technical Series Publication:**

Bojanova I (2024) Bugs Framework (BF): Formalizing Cybersecurity Weaknesses and Vulnerabilities. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP), NIST SP 800-231. https://doi.org/10.6028/NIST.SP.800-231

**NIST Author ORCID iD**

0000-0002-3198-7026

**Contact Information**

bf@nist.gov

## Abstract

The Bugs Framework (BF) is a classification of security bugs and related faults that features a formal language for the unambiguous specification of software and hardware security weaknesses and vulnerabilities. BF bugs models, multidimensional weakness and failure taxonomies, and vulnerability models define the lexis, syntax, and semantics of the BF formal language and form the basis for the definition of secure coding principles. The BF formalism supports a deeper understanding of vulnerabilities as chains of weaknesses that adhere to strict causation, propagation, and composition rules. It enables the generation of comprehensively labeled weakness and vulnerability datasets and multidimensional vulnerability classifications. It also enables the development of new algorithms for code analysis and the use of AI models and formal methods to identify bugs and detect, analyze, prioritize, and resolve or mitigate vulnerabilities.

## Keywords

bug classification; bug identification; software/hardware weakness taxonomy; vulnerability detection; safe coding; formal language; specification generation; weakness dataset; vulnerability dataset; vulnerability classification; software bug; firmware bug; hardware defect; hardware logic bug; bug triaging; software error; software fault; software weakness; hardware weakness; software vulnerability; hardware vulnerability; exploit; security failure; secure coding; vulnerability resolution; vulnerability mitigation; labeled dataset; generation tool; graph generation; AI models; formal methods; CVE; CWE; NVD; KEV.

## Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

## Audience

The intended audience includes security researchers, software and hardware developers, information technology (IT) managers, and IT executives.

# Table of Contents

## List of Figures

## 1. Introduction

The Bugs Framework (BF) [1] is a classification of security bugs and related faults with multidimensional weakness and failure taxonomies that features a formal language for the unambiguous specification of security weaknesses and vulnerabilities. The goal of BF is to help better understand and detect software, firmware, or hardware security weaknesses and vulnerabilities, as well as to resolve or mitigate them. Both cybersecurity experts and automated systems need precise descriptions of the publicly disclosed vulnerabilities and the weakness types related to them. Automated analysis via formal methods requires formal definitions of the weakness and vulnerability concepts. The automated analysis via artificial intelligence (AI) models requires comprehensively labeled weakness and vulnerability datasets.

The BF organizes bugs by the operations of orthogonal software, firmware, or hardware *execution phases*; faults by their input operands; and errors by their output results. An error either propagates to a fault or is final and enables a security failure. Bugs and faults are causes of security weaknesses, and errors and final errors are their consequences. A *bug* is a code or specification defect. A *fault* is a name, data, type, address, or size error.

A BF *weakness class* is a taxonomic category of a weakness type that relates to a distinct execution phase defined by a set of operations and their input operands and output results. It defines finite sets of ⟨*cause*, *operation*⟩→*consequence* causal relations, operation and operand attributes, and code sites. Causes are bugs or faults, and consequences are errors or final errors.

A *weakness* is an instance of a BF class with one cause, one operation, and one consequence that is expressed as a ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, or ⟨*fault*, *operation*⟩→*final error* triple and specific operation and operand attributes and sites.

A *vulnerability* is a chain of weaknesses linked by causality via a *consequence*↷*cause* propagation that eventually enables a security failure. It starts with a bug or hardware-induced fault, propagates through errors that become faults, and ends with a final error that introduces an exploit vector toward a failure. The first weakness relates to the root cause of the vulnerability, and the last relates to its sink.

BF bugs models, weakness and failure taxonomies, and vulnerability models define the BF formal language lexis, syntax, and causal semantics. The BF *bugs models* define the sets of operations for related execution phases and the proper flow between these operations. The BF *weakness taxonomies* comprise structured, orthogonal, multidimensional, and context-free BF weakness classes. The BF *failure taxonomy* comprises corresponding BF failure classes. The BF *vulnerability models* define state and specification views of a vulnerability, possibly converged and chained with other vulnerabilities. The BF formal language is generated by the BF left-to-right leftmost derivation one-symbol lookahead (LL(1)) attribute context-free grammar (ACFG) based on the BF taxonomies and models.

Analogous to the periodic table, the BF weakness taxonomies allow for the identification or prediction of as yet unencountered security weakness types, which would allow for the prediction of new kinds of vulnerabilities.

The BF taxonomies and models also form the basis for defining secure coding principles, such as input/output check safety (e.g., injection safety), memory safety (e.g., buffer overflow safety or use-after-free safety), and data type safety (e.g., floating point safety or subtype confusion safety). While the BF formal language is descriptive of weaknesses and vulnerabilities, the BF secure coding principles are prescriptive against them — they prevent bugs and faults that compromise code safety.

The BF formalism supports a deeper understanding of vulnerabilities as chains of weaknesses that adhere to strict causation, propagation, and composition rules and allows for backward bug identification from a failure. It enables a new range of research and development efforts for the creation of comprehensively labeled weakness and vulnerability datasets and the generation of formal vulnerability specifications and multidimensional vulnerability classifications.

The BF also supports the development of new static or dynamic analysis and simulation or emulation algorithms [2], as well as AI models and capabilities to identify bugs and detect vulnerabilities. Given the formal specification of code and the BF definitions of weakness and vulnerability, formal methods could also be applied to detect vulnerabilities. The next steps would be to prioritize and resolve or mitigate each of these vulnerabilities (i.e., fix the bug or a fault) to secure critical infrastructure and supply chains.

The datasets of weakness and vulnerability BF specifications formally augment the Common Weakness Enumeration (CWE) [3], the Common Vulnerabilities and Exposures (CVE) [4], and the National Vulnerability Database (NVD) [5]. However, the BF has the expressive power to clearly describe any other security weaknesses and vulnerabilities.

This NIST Special Publication (SP) provides a detailed overview of the Bugs Framework (BF) systematic approach and methodologies for classifying bugs and faults by orthogonal execution phases, formally specifying weaknesses and vulnerabilities, defining secure coding principles, generating comprehensively labeled weakness and vulnerability datasets and vulnerability classifications, and developing BF-based algorithms and systems.

Further details will be available in the following forthcoming NIST SPs:

- SP 800-231A, *Bugs Framework: Security Concepts*
- SP 800-231B, *Bugs Framework: Bugs Models*
- SPs 800-231Cx, *Bugs Framework: _yyy Taxonomy*, where _yyy is a BF class type
- SP 800-231D, *Bugs Framework: Vulnerability Models*
- SP 800-231E, *Bugs Framework: Formal Language*
- SP 800-231F, *Bugs Framework: Secure Coding Principles*
- SP 800-231G, *Bugs Framework: Tools and APIs*
- SP 800-231H, *Bugs Framework: Datasets and Classifications*
- SP 800-231I, *Bugs Framework: Systems*

## 2. Current State of the Art

The current state of the art in describing security weaknesses and vulnerabilities are the CWE [3] and CVE [4]. The current state of the art in labeling security weaknesses and vulnerabilities is the NVD [5], which assigns to a CVE the CWE weakness type that most closely matches the vulnerability. The Known Exploited Vulnerabilities (KEV) catalog [6] is also closely related to the CVE.

The CWE and CVE are widely used. The CWE is a community-developed list of software and hardware weakness types. It was developed to address "the issue of categorizing software weaknesses" and establish "acceptable definitions and descriptions of these common weaknesses" and recently added ”support for hardware weaknesses" [7]. Each CWE entry is assigned a *CWE-x* ID (identifier), where *x* is one to four digits. It provides a weakness-type description, an extended description, modes of introduction, possible mitigations, detection methods, and demonstrative examples.

The CVE is a catalog of publicly disclosed security vulnerabilities. It was initiated to address the problem of having "no common naming convention and no common enumeration of the vulnerabilities in disparate databases" [8, 9]. Each CVE entry is assigned a *CVE-yyyy-x* ID, where *yyyy* is the year of disclosure and *x* is a unique sequential number. Each CVE entry provides a vulnerability description, references to reports, and possibly links to proof of concept and code.

The CWE and CVE adopted a one-dimensional list (i.e., enumeration) approach to organizing the entries by unique IDs with natural language descriptions. The CWE added tree-based pillar, class, base, variant, and compound abstractions. Both repositories are regu-larly refined, and new weakness types, vulnerabilities, and related content are added [7].

The NVD maps CVE entries to CWE entries and assigns Common Vulnerability Scoring Sys-tem (CVSS) [10, 11] severity scores. The KEV catalog organizes publicly exploited CVEs pri-oritized for remediation, although they are not necessarily the most severe.

However, the CWE hierarchical structure implies that the weakness types are interdepen-dent and may be too broad, not orthogonal, and ambiguous. Many of the CWE and CVE descriptions are not sufficient, accurate, or precise enough [12–15]; have unclear causal-ity [16–18]; and include programming language and domain-specific notions. The CWE has gaps and overlaps in coverage [16–18], and while some gaps are being identified, new overlaps may be created [19]. Many CVEs do not describe the entire chain of weaknesses underlying the vulnerability. Some list the final error at the sink as the root cause instead of the bug or hardware-induced fault that starts the chain. Focusing on the final error helps identify mitigation techniques, but the actual root cause must be known and fixed to resolve the vulnerability. In the case of CVEs that overlap by root cause [20], fixing that one root cause would resolve all of them. These CWE and CVE challenges propagate to the NVD and KEV and may lead to imprecise or wrong CWE-to-CVE assignments by NVD.

Additionally, the CWE and CVE do not exhibit strict methodologies for tracking the weaknesses underlying a vulnerability, systematic comprehensive vulnerability labeling, or backward root cause identification from a security failure. There are no tools to aid the creation and visualization of weakness and vulnerability descriptions (see Table 1).

**Table 1.** CWE, CVE, and NVD challenges

| Repository Challenges | Imprecise Descriptions | Unclear Causality | Gaps | Overlaps | Wrong CWE Assignments | No Tracking | No Description Tools |
|---|---|---|---|---|---|---|---|
| CWE | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| CVE | ✓ | ✓ | | ✓ | | ✓ | ✓ |
| NVD | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |

The imprecise descriptions and lack of explainability make CWEs and CVEs difficult to use in modern cybersecurity research [21]. For example, the description of CWE-502 mixes the notions of validation (syntax check) and verification (semantics check), for which BF defines two distinct weakness classes [16]. The descriptions of some CWEs reveal possible causing weaknesses and even chains of weaknesses, which could be helpful but may also imply that these are the only possible causing weaknesses. They also introduce terms that are unrelated to the main weakness and may mislead experts and automated analysis about the single weakness that the CWE is meant to describe. Augmenting the CWE and CVE natural language descriptions with unambiguous formal specifications that adhere to within and between weaknesses causation rules will make them more suitable for algorithms and as comprehensively labeled datasets for training AI models [22].

Unclear causality in CVEs leads to incorrect CWE assignments. For example, in the case of CVE-2018-5907, the lack of input validation leads to integer overflow and then buffer overflow [16]. However, the NVD labels it with CWE-190: Integer Overflow or Wraparound, even though the root cause is CWE-20: Improper Input Validation. The entire chain is CWE-20→CWE-190→CWE-119, and the last one is "Improper Restriction of Operations within the Bounds of a Memory Buffer." CVE-2014-0160 Heartbleed [23] lists the final error at the sink — buffer over-read — as the root cause, while it is missing input verification that leads to pointer reposition over the upper bound and then to buffer over-read. For lack of a better match, NVD assigns the broader CWE-125, which covers both under-lower-bound and over-upper-bound reads from a buffer.

CVEs that have the same root cause are also difficult to identify. For example, the CVE-2016-7523 and CVE-2016-7524, CVE-2016-7518 and CVE-2017-6500, and CVE-2019-13295 and CVE-2019-13297 couples each have the same bug with the last couple patched in two versions of the product via two different commits [20]. As the chains are incomplete for many CVEs, there is no way to go backward from the failure to reveal the root cause.

The BF addresses the challenges with the CWE and CVE descriptions via its orthogonal, multidimensional, and context-free classification structure. The BF weakness and vulnerability specifications provide formal augmentation to the CWE, CVE, and NVD entries.

## 3. Bugs Framework Formalism

The BF is a structured multidimensional classification of security bugs and related faults as causes for the operations of distinct execution phases over their operands to result in errors and final errors as consequences. Its approach is different from the exhaustive ID-based list approach exhibited by enumerations. The BF weakness classes are organized by orthogonal sets of operations, so a BF class is identifiable by any of its operations. They allow for the expression of a weakness as a ⟨*cause*, *operation*⟩→*consequence* triple with operation and operand attributes and a vulnerability as a chain of underlying weaknesses.

A BF *weakness class* is a taxonomic representation of a weakness type defined by finite sets of operations, causes, consequences, attributes, and sites. It is associated with the operations of a distinct phase of software, firmware, or hardware execution, where weaknesses of this type could happen, as well as their input operands and output results.

A BF *operation* is the minimal input-process-output code that can produce or propagate improper data. A *cause* is a bug in the operation or a fault of an input operand. A *consequence* is an erroneous output result from the operation over the operands. The error propagates to a fault or is a final error that enables a failure. Consequently, a BF *operation* is the minimal input-process-output code that can produce an error from a bug or fault, where the error propagates to another fault or is final (i.e., it is a final error).

The *attributes* describe the operations and operands with details on what, how, and where it went wrong. They help understand the severity of the bug or fault causing the weakness. For example, pointer overbounds faults on the stack are more severe than those on the heap because buffer overflows on the stack, although easier to exploit, are more severe than those on the heap. The *sites* point to syntactic places in code that should be checked for bugs or faults that cause such weaknesses.

The BF *specification of a weakness* is based on one taxonomic BF class; it is an instance of that BF class with one cause, one operation, one consequence, and their attributes. The *operation* binds the causation within a weakness as a ⟨*cause*, *operation*⟩→*consequence* relation. For example, the *deallocation via a dangling pointer leading to a final error known as double free* is a weakness that is expressed formally via BF as ⟨*Dangling Pointer*, *Deallocate*⟩→ *Double Deallocate*. The BF *specification of a vulnerability* is a chain of such instances and their *consequence*↷*cause* between weakness propagations.

The BF is a formal system that comprises:

- Strict *definitions* of bug, fault, error, final error, weakness, vulnerability, exploit vector, and failure in the context of cybersecurity to elucidate causation and propagation rules
- *Bugs models* that define distinct execution phases with orthogonal sets of operations in which specific bugs and faults could occur and the proper flow of operations

- Structured, multidimensional, orthogonal, and context-free *weakness taxonomies* as weakness class types and a *failure taxonomy* as a failure class type
- A *vulnerability state model* as a chain of improper-state (*operation*, *operand*$_1$, …, *operand*$_n$) tuples with a bug in the operation or a fault of an operand that enables a failure
- A *vulnerability specification model* as a chain of $\langle$*cause*, *operation*$\rangle \rightarrow$*consequence* instances of BF weakness classes that ends with an instance of a BF failure class
- A *formal language* for the unambiguous causal specification of security weaknesses and vulnerabilities
- *Secure coding principles*, such as input/output check safety, memory safety, and data type safety
- *Tools* that facilitate the generation of CWE2BF and CVE2BF mappings and formal weakness and vulnerability specifications and their graphical representations
- Comprehensively labeled *weakness* and *vulnerability datasets*
- Multidimensional *vulnerability classifications* by common properties and similarities based on the BF taxonomies and secure coding principles

The BF taxonomies are structured, orthogonal, multidimensional, and context-free. *Structured* means that a weakness is expressed as a $\langle$*cause*, *operation*$\rangle \rightarrow$*consequence* triple with a precise causal relation. The transition from a weakness is expressed as an *error*$\curvearrowright$*fault* or *final error*$\curvearrowright$*exploit vector* propagation. These ensure clear causality within a weakness, between weaknesses, and for an exploit toward a failure.

*Orthogonal* means that the intersection of the sets of operations of any two BF classes is the empty set. It ensures that the BF weakness types do not overlap in coverage.

*Multidimensional* means that weaknesses are organized not only by their operations but also by their causes, consequences, and operation and operand attributes. It ensures the BF's expressive power.

*Context-free* means an operation cannot have different meanings depending on the language or domain. It ensures that the BF is applicable for code in any programming language and for any platform or application technology.

The BF formal language (see Sec. 8) is generated by the BF LL(1) ACFG, whose lexis, syntax, and semantics reflect the BF weakness taxonomies and bugs and vulnerability models that utilize the strict BF concept definitions for security bug, final error, weakness, vulnerability, exploit vector, and failure as well as fault and error. The LL1 CFG is pivotal, as it ensures precise, unambiguous specifications.

The BF bugs models and weakness taxonomies are developed iteratively according to the BF methodology (see Sec. 6.3) and alongside the BF, BFCWE, and BFCVE tools (see Sec. 10).

The BF formalism guarantees precise descriptions with clear causality of weaknesses (including CWE) and vulnerabilities (including CVE) and complete, orthogonal, and context-free weakness-type coverage. It forms the basis for the formal definition of secure coding principles, such as memory safety. It also enables the creation of comprehensively labeled weakness and vulnerability datasets, vulnerability classifications, and BF-based systems for bug identification and vulnerability detection, analysis, and resolution or mitigation.

### 3.1. BF Operation

A BF *operation* is the minimal input-process-output code that can produce or propagate improper data (see Fig. 1). An input operand or output result data is of a specific data type. A *data type* defines a set or range of data values and the operations allowed on them. It can be primitive (e.g., `char, int, double, string, boolean`) or structured (e.g., `array, record, class`). A *data value* is stored in a finite region of *memory* called an *object*. The boundaries of that memory define the *size* of the object. The *address* of the memory must be held by at least one *pointer* or determined as an offset on the stack. Otherwise, the object would be unreachable. Code (i.e., functions) and data type metadata are also stored in memory and can be referred to by pointers. A *function* is an organized, reusable block of code that takes inputs and returns outputs of specific data types.

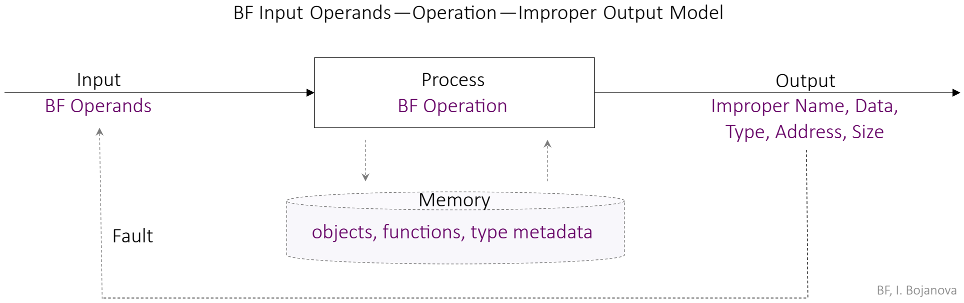

BF Input Operands—Operation—Improper Output Model
Input
BF Operands
Process
BF Operation
Output
Improper Name, Data, Type, Address, Size
Memory
objects, functions, type metadata
Fault
BF, I. Bojanova

**Fig. 1.** BF operation

Consequently, the possible entities stored in memory are objects, functions, and types. The possible BF input operands and output results are: *name*, *data* (i.e., the data value), *type* (i.e., the data type), *address*, and *size*. A BF *operation* then is the minimal input-process-output code that can produce or propagate improper *name*, *data*, *type*, *address*, or *size* (see the purple terms in Fig. 1).

BF operations could be as simple as dereferencing or repositioning a pointer or as complex as encryption or authentication involving sophisticated algorithms. Other examples include data verification, type coercion, and reading or deallocating an object. Examples of improper output results and input operands are wrong value and wrap-around as *data*, insufficient size and cast pointer as *type*, dangling pointer and over-bounds pointer as *address*, and wrong resolved object and wrong generic function bound as *name*.

The output result of a BF operation is always erroneous; for that, either the operation or an input operand is improper. An operation is improper if it has a bug, and an operand is improper if it is ill-formed (i.e., it is at fault). The erroneous output propagates to become the improper input of another BF operation or eventually is final. Consequently, a BF *operation* is the minimal input-process-output code that — because of a bug or fault — results in an error that propagates to another fault or is final (i.e., it is a final error). The fault is of a *name*, *data*, *type*, *address*, or *size* (see Fig. 1).

### 3.2. BF Bug, Fault, and Weakness

The BF bugs and faults landscape covers the operations in software, firmware, and hardware *execution phases* at appropriate levels of abstraction. The *software operations* relate to code in applications, libraries, utilities, programming languages, services, and OSs. The *firmware operations* relate to code in device drivers, basic input/output systems (BIOS), bootloaders, and microcontrollers, as well as to microcode in central processing units (CPU) and other hardware components that require low-level control and flexibility. The *hardware operations* relate to electronic circuit logic, which adheres to the same input-process-output model as software and firmware operations.

A BF *security bug* is a defect in the code or specification (i.e., metadata or algorithm) of software, firmware, or hardware circuit logic. A bug could be introduced in an operation (i.e., the improper operation) by a programmer, be the result of a design flaw, or be induced by a hardware defect. Hardware defects can result from overheating, radiation effects, electromagnetic interference (EMI), electrical noise, voltage variations, electromagnetic fields, photon injection, wear and tear, or other physical factors.

Examples of code bugs include missing code (i.e., part of an operation or an entire operation is missing) or erroneous code (e.g., use of a wrong operator in an operation). A memory bit flip due to a hardware defect can corrupt a low-level instruction. Examples of specification bugs include the use of an under-restrictive safelist for input validation or a wrong algorithm for encryption.

A bug could also resurface from design flaws, such as an unaccounted-for system configuration or environment. For example, while an operation may run perfectly in a 64-bit operating system (OS) environment, it may exhibit a security bug on a 32-bit platform. The declaration of an `int` instead of `uint` object would lead to a wraparound from a 32-bit calculation that would eventually propagate to a buffer overflow.

A BF *fault*[1] is a name, data, type, address, or size error (i.e., an improper operand). *Name* relates to a resolved or bound object, function, or data type. *Data*, *type*, *address*, and *size* relate to an object. A fault of an operand (i.e., the improper operand) could result from

[1] The IEEE defines fault as "an incorrect step, process, or data definition in a computer program" [24]. The BF differentiates bug, fault, and error as a code or specification defect of a BF operation, data-related error of an operand, and the result of an operation with a bug or faulty operand, respectively (also see Sec. 4).

a bug or another fault or be induced by a hardware defect. Only in the case of low-level storage (e.g., cache and CPU registers) is there no *type* fault.

Adding to the examples of improper input operands in Sec. 3.1, other faults include invalid data, wild pointer, wrong type, or missing overridden function. A wrong value could result from a missing validation or an erroneous calculation bug, but also from bit flips or signal disruption due to overheating or other physical factors.

The BF models a *security weakness* as an improper execution state and its transition to another weakness or a failure (see Fig. 2). An improper state is defined as an (*operation*, *operand*$_1$, $\ldots$, *operand*$_n$) tuple with at least one improper element (depicted in Fig. 2 in purple). A transition is defined by the erroneous output from the operation over its input operands. An improper operation or operand is the cause of a security weakness. The erroneous result from the operation over the operands is the consequence of that weakness and becomes a cause of another weakness or enables a failure. An operation is improper if it has a bug. An operand is improper if it is ill-formed (i.e., it is at fault).

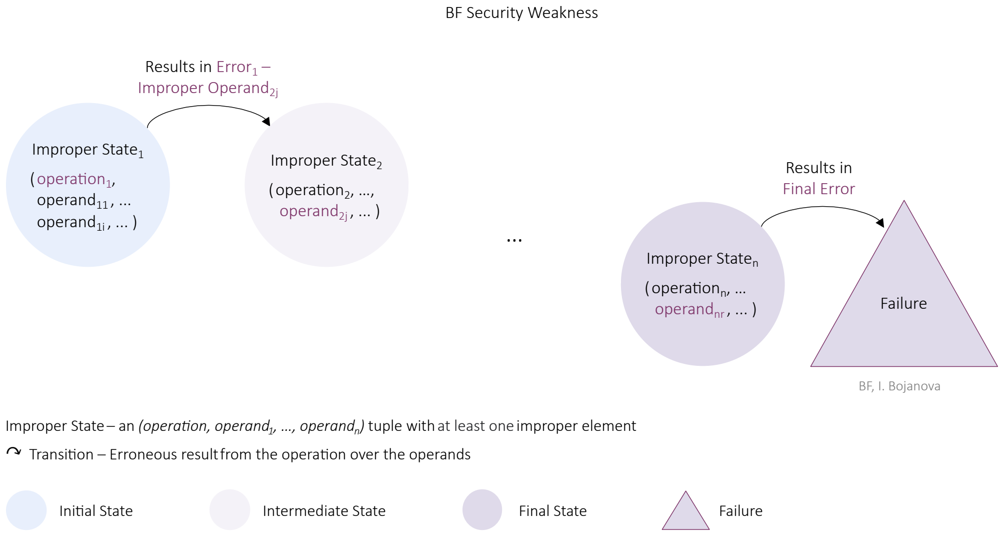

BF Security Weakness
Results in Error$_1$ – Improper Operand$_{2j}$
Improper State$_1$
(operation$_1$, operand$_{11}$, ... operand$_{1i}$, ... )
Improper State$_2$
(operation$_2$, ..., operand$_{2j}$, ... )
...
Results in Final Error
Improper State$_n$
(operation$_n$, ... operand$_{nr}$, ... )
Failure
BF, I. Bojanova
Improper State – an (operation, operand$_1$, ..., operand$_n$) tuple with at least one improper element
↷ Transition – Erroneous result from the operation over the operands
Initial State
Intermediate State
Final State
Failure

**Fig. 2.** BF security weakness

The initial state (depicted in Fig. 2 in blue) is caused by a bug that, if fixed, will resolve the weakness. An intermediate state (in light purple) is caused by a fault. The final state (in dark purple) results in an undefined or exploitable system behavior (i.e., a final error). For example, in Fig. 2, the improper *operation*$_1$ from *Improper State*$_1$ results in improper *operand*$_{2i}$ that causes *Improper State*$_2$. The last *operation*$_n$ results in a *Final Error* that enables a *Failure*.

The possible improper states depending on the cause and consequence of a security weakness are presented in Fig. 3. An improper state is caused by a security bug (i.e., the operation is improper) or a fault (i.e., an input operand is improper). The consequence of an improper state is an error that propagates to another fault or is a security final error.

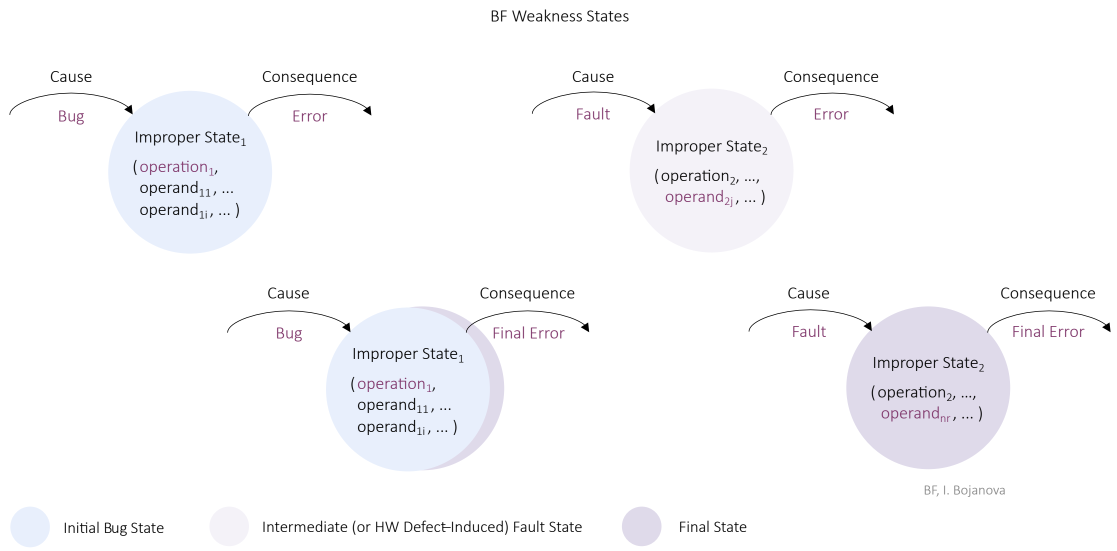


**Fig. 3.** BF weakness states

A BF *security weakness* is a ⟨*cause*, *operation*⟩→*consequence* relation triple, which is formally a ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, or ⟨*fault*, *operation*⟩→*final error* causal triple of a bug or fault weakness type. A bug informs that the operation is improper, while a fault informs about an improper operand.

Examples of weaknesses include ⟨*Missing Code*, *Sanitize*⟩→*SQL Injection*, ⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer*, and ⟨*Dangling Pointer, Read*⟩→*Use After Deallocate*. In the C programming language, the last final error is known as use after free.

### 3.3. BF Vulnerability

The BF models a *security vulnerability* as a chain of improper states that propagate as the *error* (i.e., the erroneous output) from one state becomes the *fault* (i.e., the improper input) for the next state until a *final error* that can be exploited toward a security *failure* is reached (see Fig. 4). That is, a vulnerability is a causal chain of weaknesses. The initial state (depicted in blue) is caused by a software or firmware *bug* (i.e., an operation defect) (see the blue solid arrow), which if fixed will resolve the vulnerability. A vulnerability chain may also start from a hardware defect-induced *bug* or *fault* (see the green dashed arrows), which if fixed will resolve that vulnerability.

A propagation state (depicted in light purple) is caused by a *fault* (i.e., an operand error). The final state (shown in dark purple) results in a *final error* (i.e., an undefined or exploitable system behavior) and can lead to a *failure* (i.e., a violation of a system security requirement). It usually directly relates to a CWE, but there are also CWEs that correspond to initial or propagation weakness states. An *error* is the result of an improper state operation over its operands. It becomes an improper operand — a *fault* — for the next improper state. A *final error* is the result of the operation from the final improper state. It introduces an *exploit vector* — that is, the pathway for exploitation — toward a security failure.

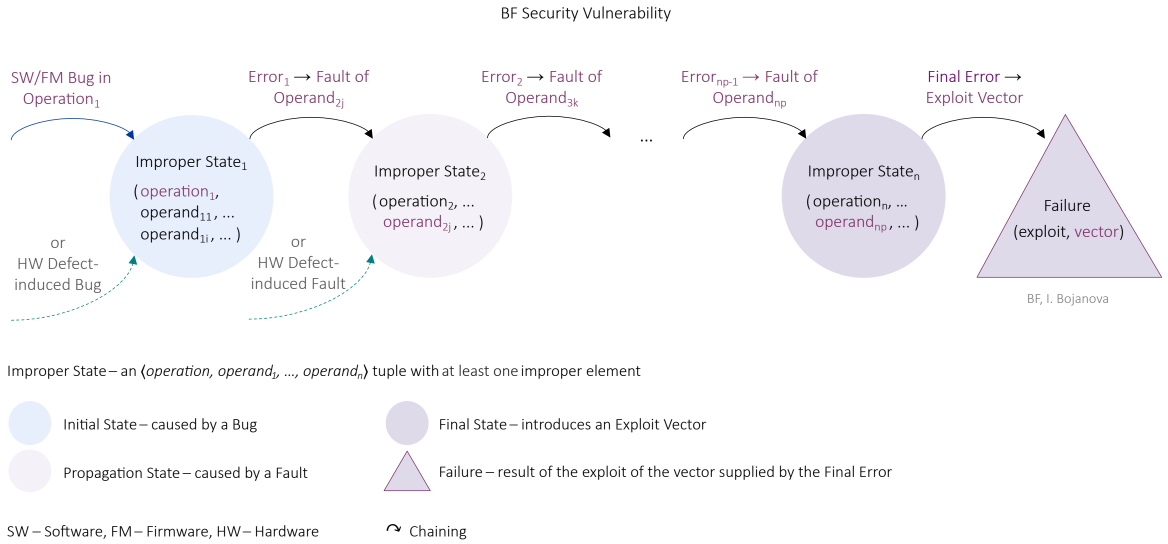

BF Security Vulnerability
SW/FM Bug in Operation$_1$
Error$_1$ → Fault of Operand$_{2j}$
Error$_2$ → Fault of Operand$_{3k}$
...
Error$_{np-1}$ → Fault of Operand$_{np}$
Final Error → Exploit Vector
Improper State$_1$
( operation$_1$, operand$_{11}$, ... operand$_{1i}$, ... )
or HW Defect-induced Bug
Improper State$_2$
( operation$_2$, ... operand$_{2j}$, ... )
or HW Defect-induced Fault
Improper State$_n$
( operation$_n$, ... operand$_{np}$, ... )
Failure
(exploit, vector)
BF, I. Bojanova
Improper State – an ⟨operation, operand$_1$, ..., operand$_n$⟩ tuple with at least one improper element
Initial State – caused by a Bug
Propagation State – caused by a Fault
Final State – introduces an Exploit Vector
Failure – result of the exploit of the vector supplied by the Final Error
SW – Software, FM – Firmware, HW – Hardware
Chaining

**Fig. 4.** BF security vulnerability

For example, in Fig. 4, *operation*$_1$ from *Improper State*$_1$ is improper due to a bug and results in *Error*$_1$ that propagates to the improper *operand*$_{2j}$, which leads to *Improper State*$_2$. The last *operation*$_n$ with improper *operand*$_{np}$ results in a *Final Error* that propagates to an *Exploit Vector* to enable a *Failure*.

As errors propagate to faults, the examples of faults also apply to errors. However, an error may be on a higher level of abstraction than a specific fault. For example, an inconsistent value error may propagate to the more specific wrong argument data fault (e.g., see the discussion about Fig. 5) or wrong size fault (e.g., see the discussion about Fig. 17).

Examples of final errors include integer overflow, query injection, buffer overflow, and side communication channels. Legitimate and side channels may also be direct exploit vectors without prior causation. Examples of failures include information exposure (IEX) (i.e., confidentiality loss), data tampering (TPR) (i.e., integrity loss), denial of service (DOS) (i.e., availability loss), and arbitrary code execution (ACE) (i.e., everything could be lost).

The initial *bug* state is of an improper operation over proper operands. It is the state with a defect in the operation. The bug must be fixed to resolve the vulnerability. A *fault* state is of a proper operation over an improper operand. It is a state with a defect in an operand that — if fixed — would only mitigate a vulnerability. If the initial state is caused by a hardware defect-induced fault, its fix will resolve the vulnerability.

Vulnerabilities may also converge at their final states and chain via faults resulting from exploits. The converged final states enable a security failure, which would not have been harmful if only one were present. The chained faults-only vulnerabilities propagate toward a final security failure.

A particular BF security vulnerability chain may correspond to more than one CVE. For example, BadAlloc is a vulnerability pattern that covers more than 25 similar CVEs [25] related to memory allocators, such as `malloc()` and `calloc()`. They were found in widely used real-time operating systems (RTOS), standard C libraries, IoT device SDKs, and other self-memory management applications going as far back as the early ‘90s [26].

The BF causal chain of the BadAlloc vulnerability pattern comprises five BF weaknesses [1]: *DVR*↷*TCM*↷*MMN*↷*MAD*↷*MUS* (see Fig. 5). The first weakness is at the input data verification phase of software execution. The memory allocation implementation has no proper size verification toward the maximum allowed value accounting for how the requested memory size is calculated — that is, a BF Data Verification (DVR) weakness [1]. This input becomes a wrong argument for a calculation that produces a value greater than the maximum integer allowed for the particular operating environment (e.g., $2^{32} - 1$ for a 32-bit OS) and wraps around the result (i.e., integer overflow error) — that is, a BF Type Computation (TCM) weakness [1].

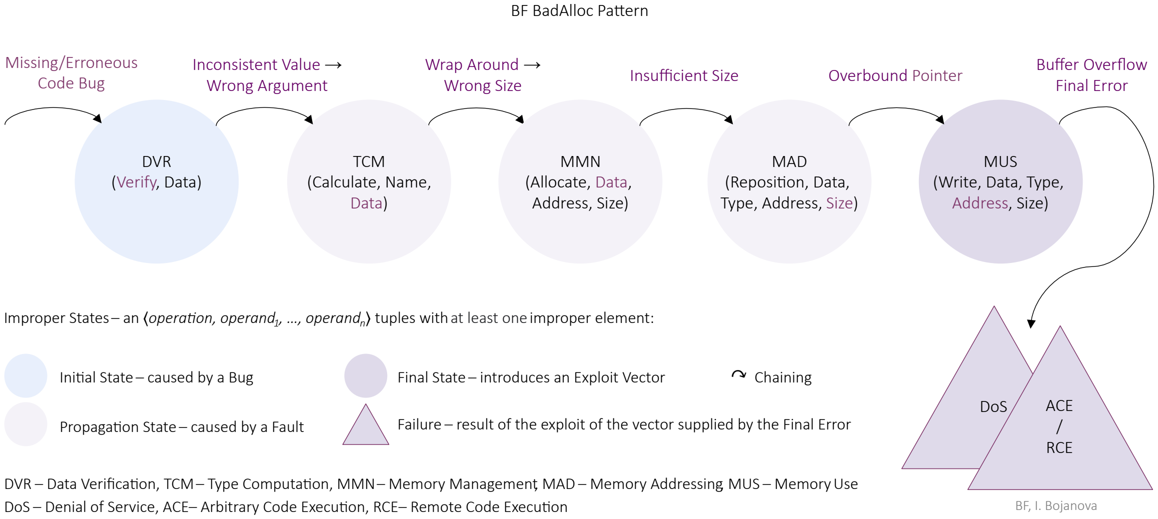

BF BadAlloc Pattern
Missing/Erroneous Code Bug
Inconsistent Value → Wrong Argument
Wrap Around → Wrong Size
Insufficient Size
Overbound Pointer
Buffer Overflow Final Error
DVR (Verify, Data)
TCM (Calculate, Name, Data)
MMN (Allocate, Data, Address, Size)
MAD (Reposition, Data, Type, Address, Size)
MUS (Write, Data, Type, Address, Size)
DoS
ACE / RCE
Improper States – an (operation, operand1, ..., operandn) tuples with at least one improper element:
Initial State – caused by a Bug
Propagation State – caused by a Fault
Final State – introduces an Exploit Vector
Failure – result of the exploit of the vector supplied by the Final Error
↷ Chaining
DVR – Data Verification, TCM – Type Computation, MMN – Memory Management, MAD – Memory Addressing, MUS – Memory Use
DoS – Denial of Service, ACE – Arbitrary Code Execution, RCE – Remote Code Execution
BF, I. Bojanova

**Fig. 5.** BF BadAlloc pattern

Consequently, a much smaller wrong size is used at allocation resulting in a memory buffer with an insufficient size — that is, a BF Memory Management (MMN) weakness [1]. This allows a pointer to be repositioned outside the buffer boundary — a BF Memory Addressing (MAD) weakness [1]— and data to be written there — a BF Memory Use (MUS) weakness [1]. The buffer overflow final error can then be exploited toward a DOS or ACE (specifically, remote code execution [RCE] on a targeted device) failure.

CVE-2021-21834 is one particular vulnerability that strictly follows the BadAlloc pattern (see its BF CVE-2021-21834 specification at [1]).

### 3.4. BF Bug Identification

If there is a cybersecurity failure, there should be a way to identify the root cause (i.e., the security bug) and the chain of triggered improper execution states that enables the failure.

Theoretically, addressing the problem of identifying a security bug would be first to generate the graph of all possible vulnerability chains of weaknesses. Then, search the graph via brute force recursive backtracking with specific constraints to find the set of possible valid paths. Finally, select the only proper path via code analysis.

However, the BF formalism ensures predictive recursive-descent parsing that does not require backtracking, as the BF formal language is generated by an LL(1) CFG. Knowing the failure and the possible transitions at execution that adhere to the BF causation within a weakness as a ⟨*cause*, *operation*⟩→*consequence* relation and between weaknesses via a *consequence*↷*cause* propagation (see Sec. 7.2), the bug can be identified going backward ↶ from the *final weakness* until an *operation* is improper (see Fig. 6). Fixing the bug within that operation would resolve the vulnerability.

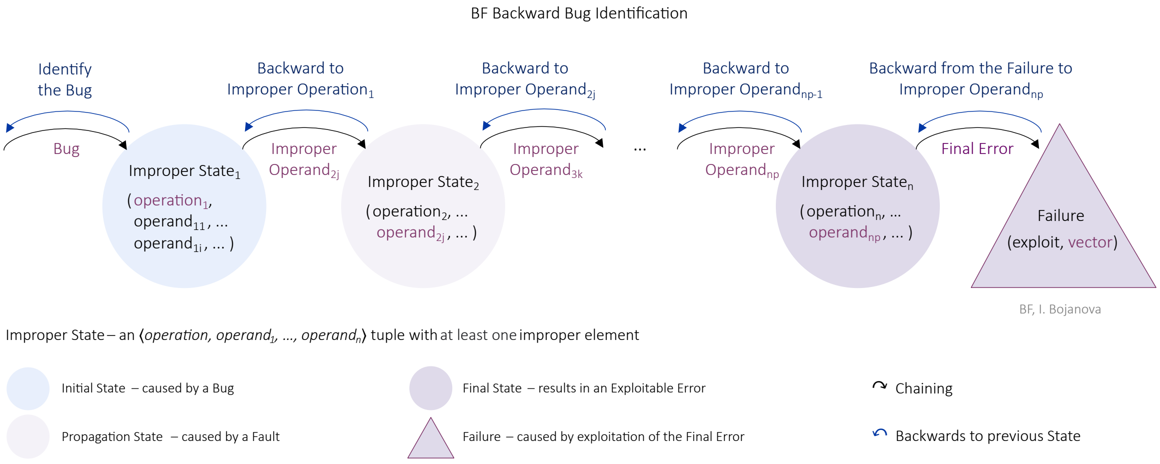

BF Backward Bug Identification
Identify the Bug
Backward to Improper Operation$_1$
Backward to Improper Operand$_{2j}$
Backward to Improper Operand$_{np-1}$
Backward from the Failure to Improper Operand$_{np}$
Bug
Improper Operand$_{2j}$
Improper Operand$_{3k}$
...
Improper Operand$_{np}$
Final Error
Improper State$_1$
(operation$_1$, operand$_{11}$, ... operand$_{1i}$, ... )
Improper State$_2$
(operation$_2$, ... operand$_{2j}$, ... )
Improper State$_n$
(operation$_n$, ... operand$_{np}$, ... )
Failure
(exploit, vector)
BF, I. Bojanova
Improper State – an ⟨*operation, operand$_1$, ..., operand$_n$*⟩ tuple with at least one improper element
Initial State – caused by a Bug
Propagation State – caused by a Fault
Final State – results in an Exploitable Error
Failure – caused by exploitation of the Final Error
↷ Chaining
↶ Backwards to previous State

**Fig. 6.** BF backward bug identification

Using the BF formal language syntax and semantics that are based on the BF taxonomies, models, and causation and propagation rules, (see Sec. 5, 6, 7, and 8) a state tree can be directly generated backward starting from a failure and a final error or weakness. The state tree is an undirected graph with exactly one simple path between any pair of nodes. The failure is the root of the tree, and each path is a reverted possible vulnerability specification chain of weaknesses from the final error through faults to a bug. A weakness is specified as a ⟨*bug/fault*, *operation*⟩→*error/final error* causal triple.

This methodology allows for the generation of a reasonable number of possible BF specification chains of weaknesses for a particular CVE. The only proper path can then be identified as the rest get eliminated via code analysis.

## 4. BF Security Concepts

> A BF *security bug* or *weakness type* relates to a distinct software, firmware (including microcode), or hardware circuit logic *execution phase* defined by a set of BF operations and their input operands and output results.
>
> A BF *operation* is the minimal input-process-output code that can produce or propagate an improper *name*, *data*, *type*, *address*, or *size*.

The BF defines the concepts of bug, fault, error, final error, weakness, vulnerability, exploit vector, and failure in the context of cybersecurity to provide the level of detail and granularity needed to understand the causation within a weakness and the causation and propagation between weaknesses and between vulnerabilities.

- A *security bug* is a code or specification defect (i.e., an operation defect) in software, firmware, or hardware circuit logic — that is, proper operands over an improper operation. The specification includes the operation metadata and algorithm.

  A bug could be introduced by a programmer, be the result of a design flaw, or induced by a hardware defect (e.g., due to overheating). A bug could also resurface from a design flaw (e.g., an unaccounted-for system configuration or environment).

- A *fault* is a name, data, type, address, or size error (i.e., an operand error) — that is, an improper operand over a proper operation.

  A fault could result from a bug or another fault or be induced by a hardware defect. In the case of low-level storage (e.g., cache and CPU registers), there is no *type* fault.

- An *error* is the result of an operation with a bug or faulty operand that propagates to a fault of an operand of another operation.

- A *security final error* is an undefined or exploitable system behavior. A final error results from an operation with a bug or faulty operand.

- A *security weakness* is a ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, or ⟨*fault*, *operation*⟩→*final error* causal triple.

- A *security vulnerability* is a causal chain of weaknesses that starts with a bug or hardware defect-induced fault, propagates through errors that become faults, and ends with a final error that introduces an exploit vector.

  The first weakness concerns the root cause of the vulnerability, and the last weakness concerns its sink.

- A *security exploit vector* is the pathway for the exploitation of a vulnerability.

- A *security failure* is a violation of a system security requirement caused by the exploitation of a security vulnerability.

The BF security concept definitions are contextually visualized in Fig. 7. Following the blue solid initial arrow, a security vulnerability may start with a software or firmware *security bug* (i.e., a code or specification defect within an operation). Following the green dashed arrow, a vulnerability chain may also start from a hardware defect-induced *fault*.

Fixing the bug or hardware defect-induced fault will resolve the vulnerability, as well as any other vulnerability with the same root cause. Fixing a propagated fault, including the cause of the final error at the sink, will only mitigate the vulnerability. Occasionally, several vulnerabilities must converge at their final errors for an exploit to be harmful. Fixing the bug or starting fault of at least one of the chains would avoid the failure. An exploit of a vulnerability may result in a fault starting a new faults-only vulnerability. Fixing the bug or starting fault of the first vulnerability will resolve the entire chain of vulnerabilities.

For more details, refer to the forthcoming SP 800-231A, *Bugs Framework: Security Concepts*.

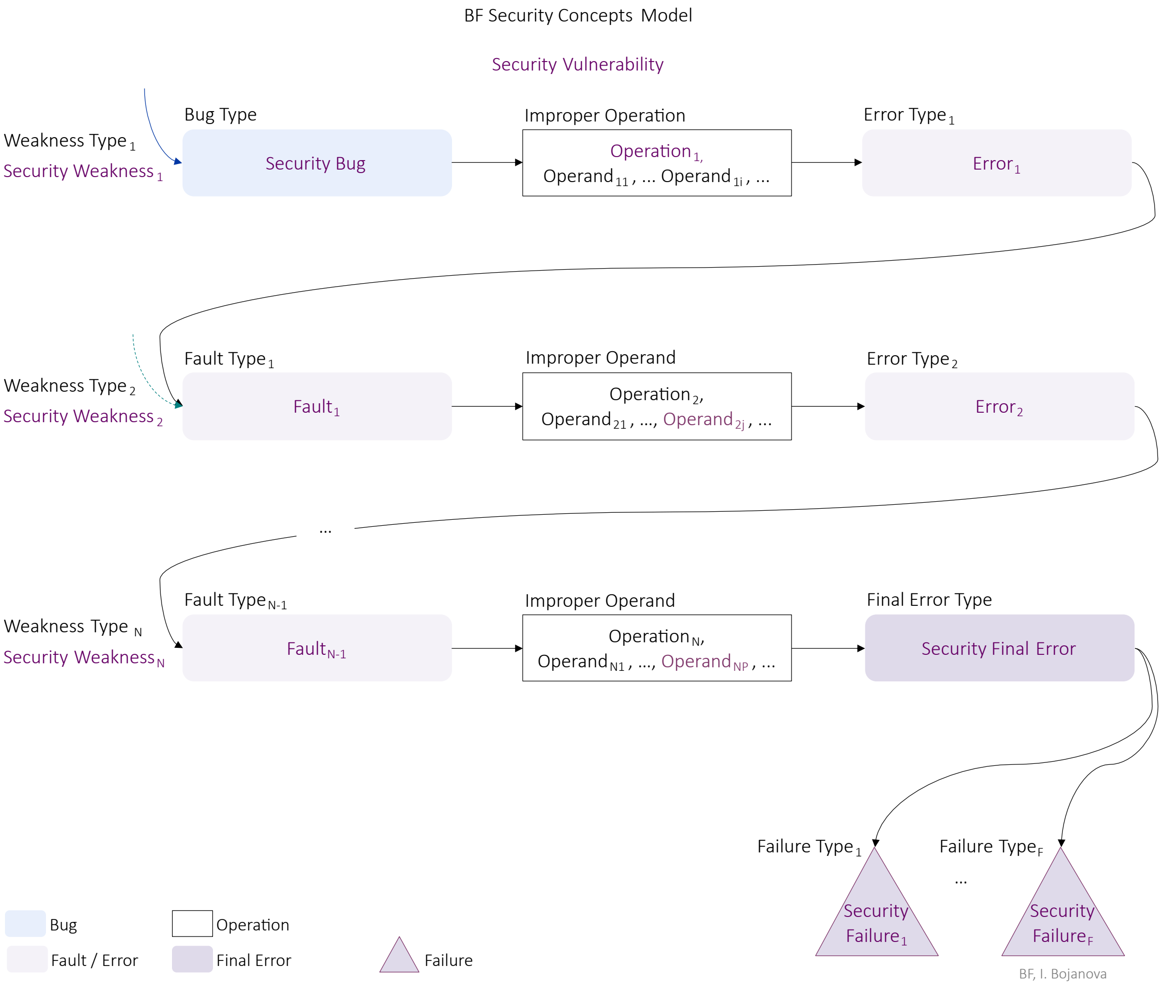


**Fig. 7.** BF security concepts

## 5. BF Bugs Models

The BF security bugs and related faults landscape covers the operations (i.e., the BF operations) in software, firmware, and hardware execution phases at appropriate levels of abstraction. A BF *operation* is the minimal input-process-output code that — because of a bug or fault — results in an error that propagates to another fault or is final (see Sec. 3.1).

The BF *bugs models* define related execution phases with orthogonal sets of operations in which particular types of bugs or faults could occur. They also define the proper flow of operations within and between the phases, which helps identify causation between weaknesses, as well as missing operations (i.e., missing code bugs) backward from a failure.

Some execution phases may only be on an application level (e.g., input/output check), while others may cover deeper levels of abstraction (e.g., the programming language type system, the OS file system, or the CPU). In any case, if there is a security failure, there must have been an operation with a security bug or a hardware defect-induced fault that propagated through faults of other operations until a security final error that introduces an exploit vector is reached.

### 5.1. BF Input/Output Check (_INP) Bugs Model

The BF Input/Output Check (_INP) Bugs Model shows that input/output data check bugs could be introduced at the *data validation* (DVL) or *data verification* (DVR) execution phase (see Fig. 8). The phases determine the BF _INP classes: Data Validation (DVL) and Data Verification (DVR) [1, 16].

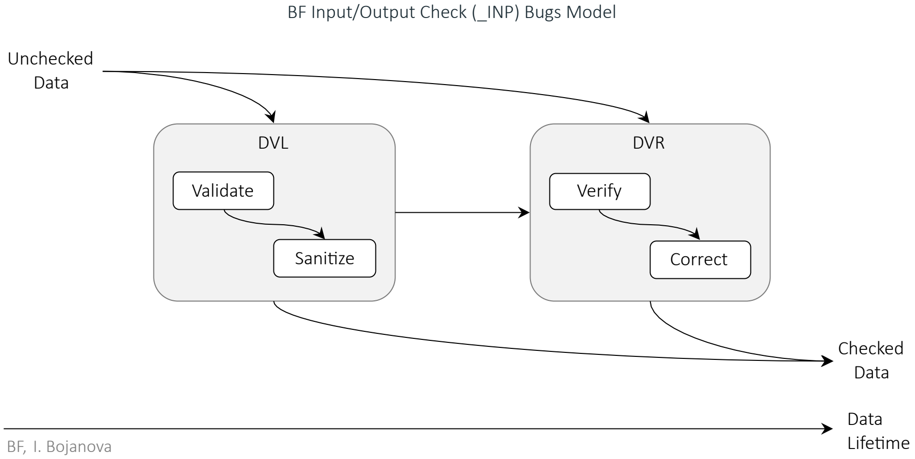


**Fig. 8.** BF Input/Output Check (_INP) Bugs Model

Each input/output check-related bug or fault involves a *Validate*, *Sanitize*, *Verify*, or *Correct* operation. According to the flow of operations, input/output data must be validated and sanitized and/or verified and corrected.

## 5.2. BF Memory (_MEM) Bugs Model

The BF Memory (_MEM) Bugs Model shows that memory-related bugs could be introduced at any phase in the life cycle of an object: *memory addressing* (MAD), *memory allocation* (MAL), *memory use* (MUS), or *memory deallocation* (MDL) (see Fig. 9). The phases determine the BF _MEM classes: Memory Addressing (MAD), Memory Management (MMN) that combines the MAL and MDL phases, and Memory Use (MUS) [1, 17].

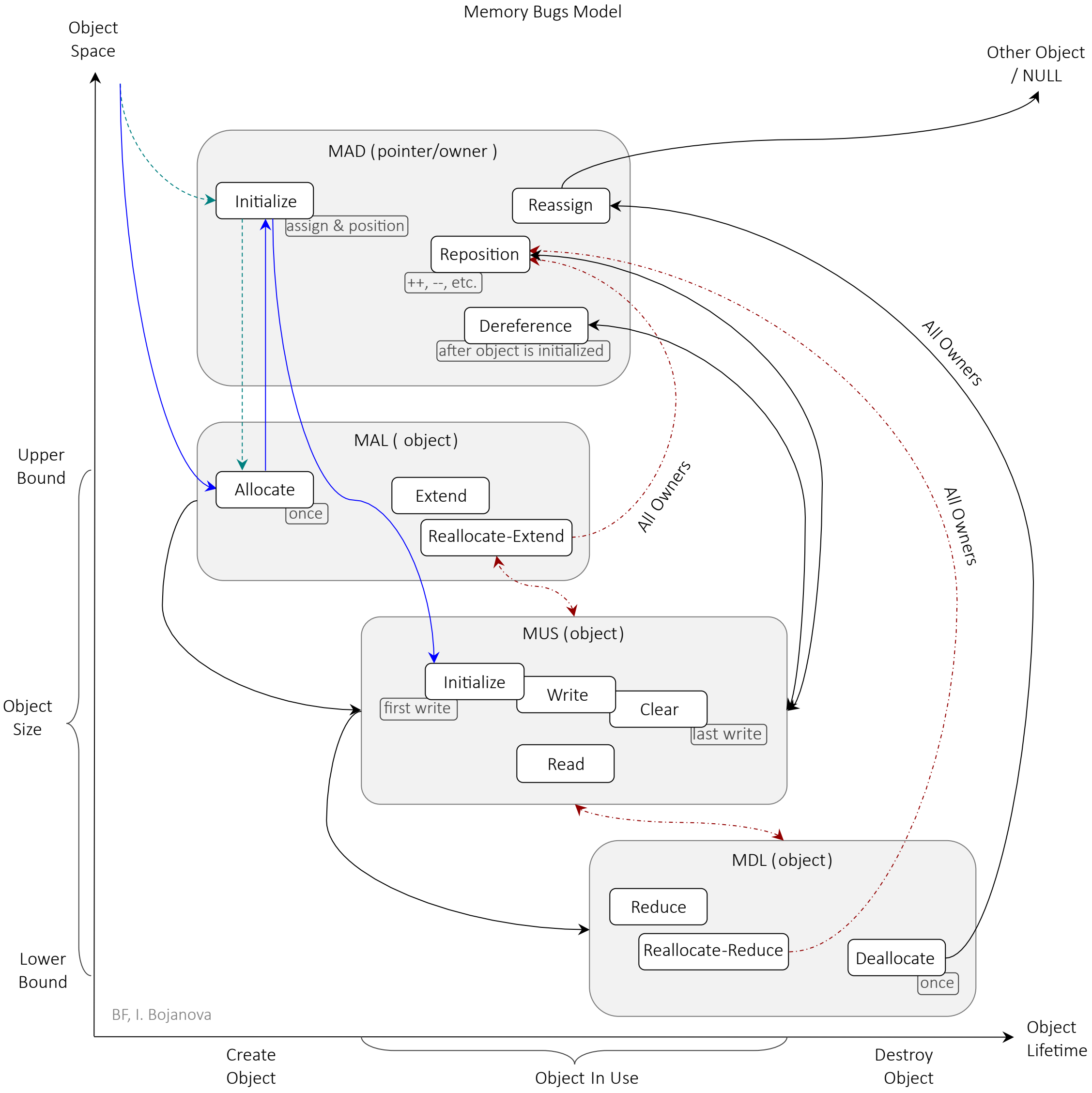


**Fig. 9.** BF Memory (_MEM) Bugs Model

Each memory-related bug or fault involves an *Initialize Pointer*, *Dereference*, *Reposition*, *Reassign*, *Allocate*, *Extend*, *Reallocate–Extend*, *Initialize Object*, *Read*, *Write*, *Clear*, *Reduce*, *Reallocate–Reduce*, or *Deallocate* operation.

The main memory-related operations flow is presented in Fig. 9 via blue and black solid arrows. The green dashed arrows show the flow for allocation at a specific address. The red dot-dashed arrows show the extra flow in case of reallocation. Following the blue arrows, the first operation is MAL *Allocate* an object. Following the green arrows, the first operation is MAD *Initialize Pointer*. The next operation following the blue arrows must be MAD *Initialize Pointer* for the allocated object to the address returned by the *Allocate* operation. In contrast, the next operation following the green arrows must be MAL *Allocate* an object at the address that the pointer holds.

After an object is allocated and its pointer initialized, MUS *Initialize Object* (i.e., the first write) must follow. Then, it may be accessed via MAD *Dereference* and used via MUS *Read* or *Write* at any point before it is cleared and deallocated.

The boundaries and size of an object set at allocation can be changed via MAL *Extend*, MAL *Reallocate-Extend*, MDL *Reduce*, or MDL *Reallocate-Reduce*. Operations that involve reallocation must be followed by MAD *Reposition* for all of the pointers that own the object. MDL *Deallocate* an object must be preceded by MUS *Clear* (i.e., the last write) and followed by MAD *Reassign* for all of its pointers to either $NULL$ or another object.

### 5.3. BF Data Type (_DAT) Bugs Model

BF Data Type (_DAT) Bugs Model shows that data type bugs could be introduced at the *declaration* (DCL), *name resolution* (NRS), *data type conversion* (TCV), or *data type-related computation* (TCM) execution phase (see Fig. 10). The phases determine the BF _DAT classes: Declaration (DCL), Name Resolution (NRS), Type Conversion (TCV), or Type Computation (TCM) [1, 18]. Each data type-related bug or fault involves a *Declare*, *Define*, *Refer*, *Call*, *Cast*, *Coerce*, *Calculate*, or *Evaluate* operation.

According to the data type-related operations flow shown in Fig. 10, the first operations over an entity (i.e., object, function, data type, or namespace) are DCL *Declare* and DCL *Define*. Then, it can be referred to in code by its name via NRS *Refer*. Names that are referred to in remote scopes get resolved via namespaces. Resolved data types get bound to objects, functions, or generic data types according to their declarations (see the purple dot-dashed arrow flow). Resolved functions get bound to implementations and may be called via NRS *Call*.

A resolved and bound object may be explicitly converted to another data type via TCV *Cast* and used to call a member function via NRS *Call* or as an argument or return of a computation function. A passed-in argument is expected to be of the declared parameter data type, and the passed-out result is expected to be of the return data type. Otherwise, TCV *Cast* is expected before or at the end of the call (see the blue large-dashed arrow flow), or the value will get implicitly converted via TCV *Coerce* to the parameter or return data type, respectively (see the green dashed arrow flow).

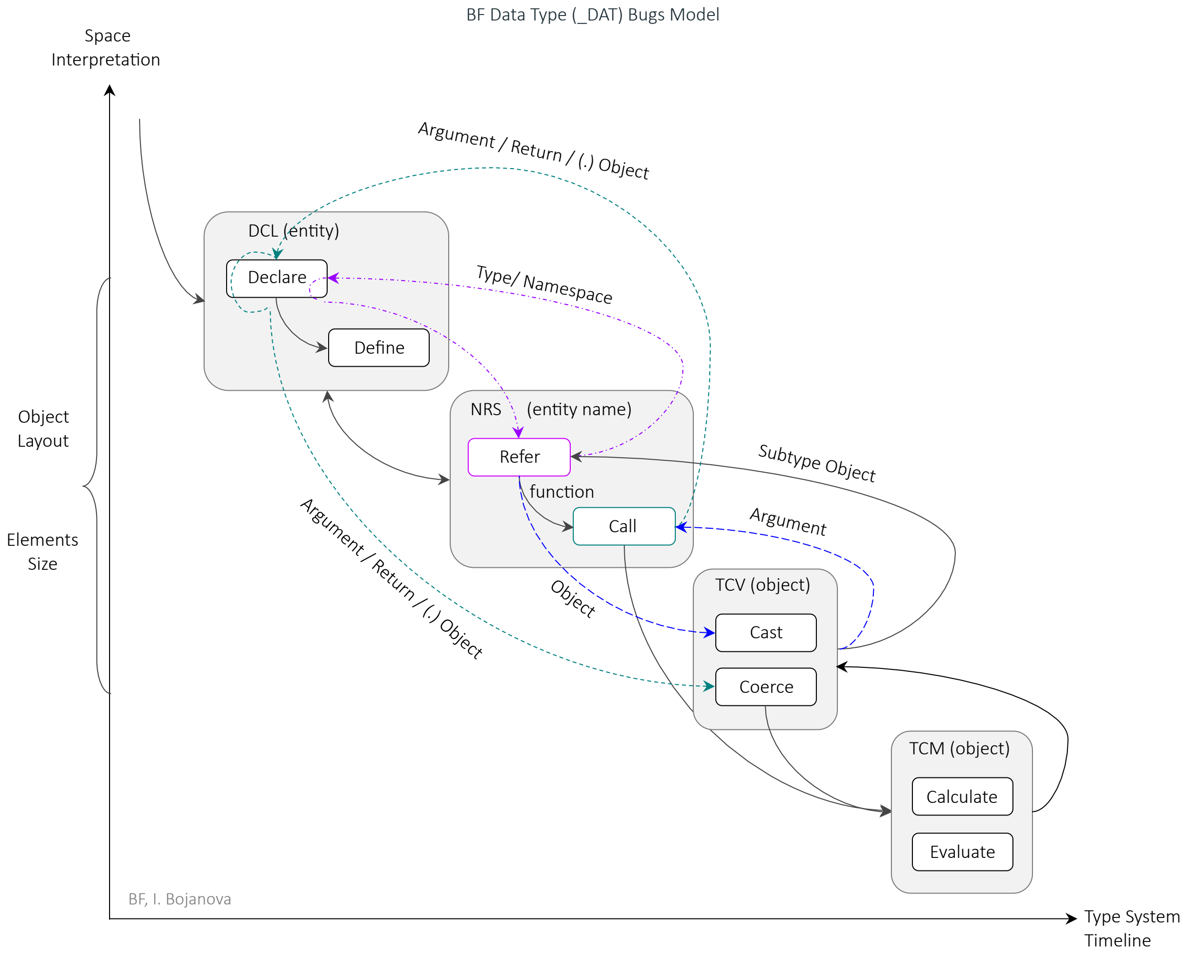


**Fig. 10.** BF Data Type (_DAT) Bugs Model

The green arrow flow is only about passed-in or passed-out objects that are coerced. It starts from NRS *Call* and never from DCL *Declare*.

A comprehensive BF bugs model would combine and connect all BF bugs models via the proper flow between their operations. For example, _DAT DCL *Declare* and _INP DVR *Verify* may be followed by _MEM MAD *Reposition*, _DAT TCM *Calculate*, or _DAT TCV *Coerse*. _DAT TCV *Coerse* may be followed by _MEM MMN *Allocate*, _MEM MAD *Reposition*, _DAT TCM *Calculate*, or _MEM MMN *Deallocate*. _DAT TCM *Calculate* may be followed by _MEM MMN *Reallocate-Reduce*.

For more bugs models and details, refer to the forthcoming SP 800-231B, *Bugs Framework: Bugs Models*.

## 6. BF Taxonomy

The BF taxonomy comprises weakness and failure categories. The BF *Weakness* category comprises BF weakness class types, such as:

- BF Input/Output Check(_INP) class type — Weaknesses that lead to input/output data check-related errors or introduce injection exploit vectors
- BF Memory (_MEM) class type — Weaknesses that lead to memory-related errors or introduce memory corruption/disclosure exploit vectors
- BF Data Type (_DAT) class type — Weaknesses that lead to data type-related errors or introduce type compute exploit vectors

The BF *Failure* category comprises the BF failure class type:

- BF *Failure (_FLR)* class type — Failures that lead to the loss of a security property due to the exploit of a vulnerability

### 6.1. BF Weakness Classes

The BF weakness taxonomy structure is based on orthogonal by operations phases of software, firmware, and hardware execution. A BF weakness class defines sets of possible bugs and faults as causes for the operations of a specific phase over their operands to result in errors and final errors as consequences.

As an error propagates to a fault (see Sec. 4), the set of errors is the same as the set of faults across classes. However, a specific propagation may be via values on different levels of abstraction (see Sec. 3.2). Similarly, the set of final errors across classes is the same as the set of exploit vectors toward failures. A BF weakness class also defines operation and operand attributes and code sites.

A BF weakness class type encompasses strictly defined weakness classes of closely related execution phases. For example, the BF _INP class type comprises the Data Validation (DVL) and Data Verification (DVR) classes. The BF _MEM class type comprises the Memory Addressing (MAD), Memory Management (MMN), and Memory Use (MUS) classes [17]. The BF _DAT class type comprises the Declaration (DCL), Name Resolution (NRS), Type Conversion (TCV), and Type Computation (TCM) classes [18]. For all current BF class types, refer to the BF Taxonomy at at [1].

The definition of the BF Data Validation (DVL) class (see Fig. 11) is "Data is validated (i.e., syntax check) or sanitized (i.e., escape, filter, or repair) improperly." It organizes security bugs by the *Validate* and *Sanitize* operations and faults by their *Data* operand as causes [16]. Possible causes are the *Missing Code* bug and *Corrupted Policy Data* fault. Possible consequences are the *Invalid Data* error and *Query Injection* and *Command Injection* final injection errors, which relate to input/output check safety (see 9.1).

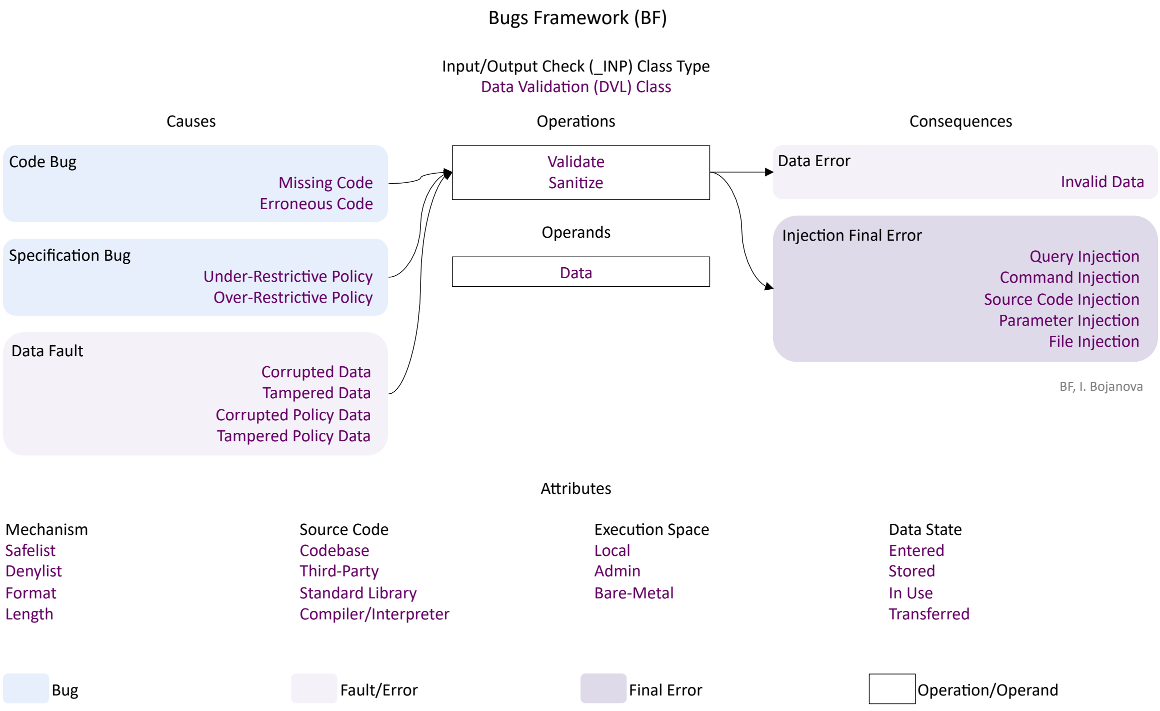


**Fig. 11.** BF Data Validation (DVL) class

The definition of the BF Memory Use (MUS) class (see Fig. 12) is "An object is initialized, read, written, or cleared improperly." It organizes security bugs by the *Initialize Object*, *Read*, *Write*, and *Clear* operations and faults by their *Data*, *Type*, *Address*, and *Size* operands as causes [17]. Possible causes are the *Wrong Size* and *Cast Pointer* faults. Possible consequences are the *Uninitialized Object* error and *Buffer Overflow* and *Use After Deallocate* (e.g., use after free or use after return) memory corruption or disclosure final errors, which relate to memory safety (see 9.2).

The definition of the BF Type Conversion (TCV) class (see Fig. 13) is "Data is converted or coerced into other types improperly." It organizes security bugs by the *Cast* and *Coerce* operations and faults by their *Name*, *Data*, and *Type* operands as causes [18]. Possible causes are the *Over Range* and *Wrong Type* faults. Possible consequences are the *Rounded Value* and *Downcast Pointer* errors, which relate to data type safety (see 9.3).

The BF strictly defines the type taxons (e.g., see the terms in black in Figs. 11, 12, and 13) for causes as bugs or faults, for consequences as errors or final errors, and for operation and operand attributes. For example, the *Specification Bug*, *Data Fault*, *Injection* and *Memory Corruption/Disclosure* Final Errors (see Figs. 11 and 12) taxon types are defined in Table 2.

The BF also strictly defines the value taxons (e.g., see the terms in purple in Figs. 11, 12, and 13) for class, operations, causes (as bugs or faults), consequences (as errors or final errors), and operation and operand attributes. For example, the *Under-Restrictive Policy* bug,

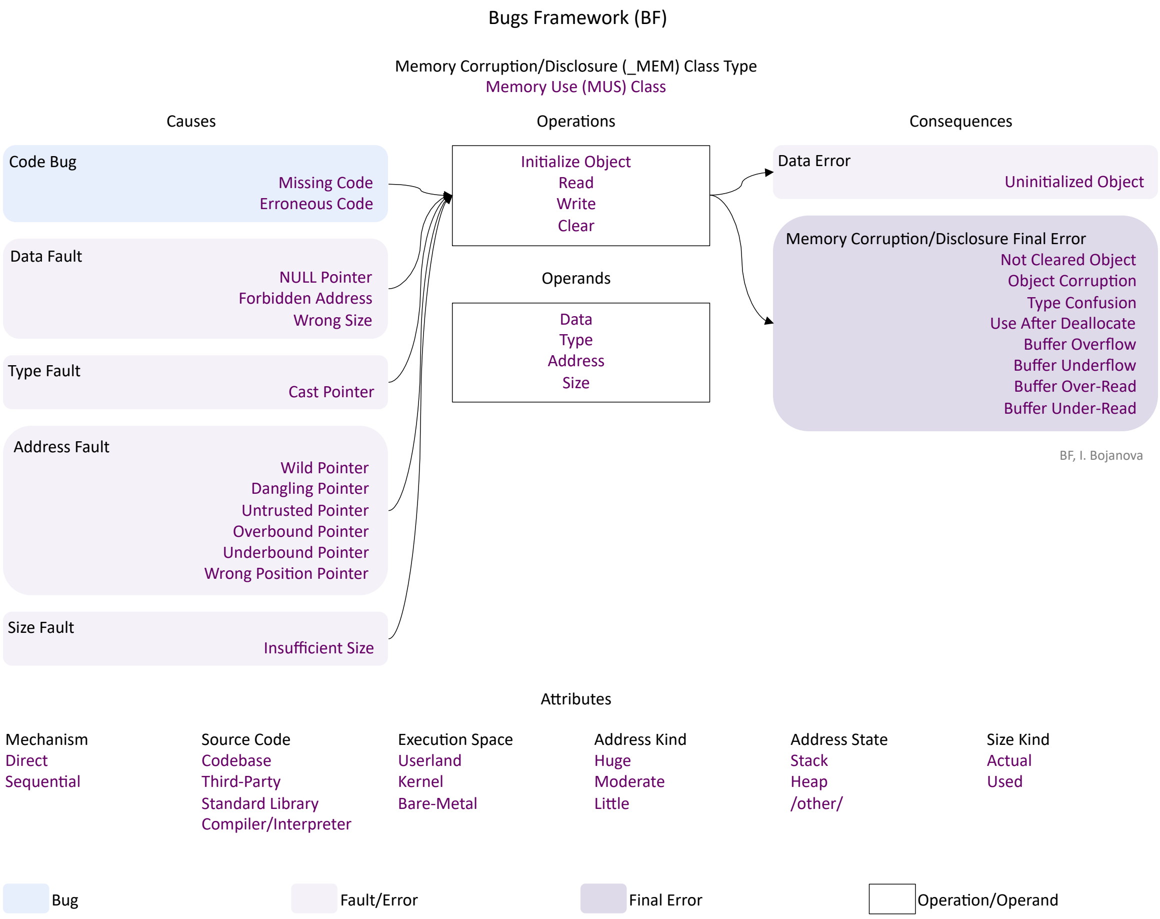


**Fig. 12.** BF Memory Use (MUS) class

*Wrong Size* fault, *Sanitize* and *Write* operations, and *Query Injection* and *Buffer Overflow* final errors (see Figs. 11 and 12) taxon values are defined in Table 3.

The operation attribute types are defined in Table 4. Their values per BF class may be different for the same operation attribute type (e.g., the *Mechanism* for MUS *Write* is *Direct* or *Sequential*, while for TCV *Coerce*, it is *Pass in* or *Pass out*). The possible operand attribute types are defined in Table 5. Their values per BF class may be different for the same operand attribute type.

Each BF class taxonomy defines a matrix of semantic rules for causation within a weakness, as some combinations of its cause, operation, and consequence value taxons may not be meaningful. They are expressed as ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, and ⟨*fault*, *operation*⟩→*final error* triples. For example, ⟨*Wrong Size*, *Write*⟩→*Buffer Overflow* is a valid triple, while ⟨*Wrong Size*, *Write*⟩→*Buffer Over-Read* is not, as the operation is *Write* but the final error is about reading.

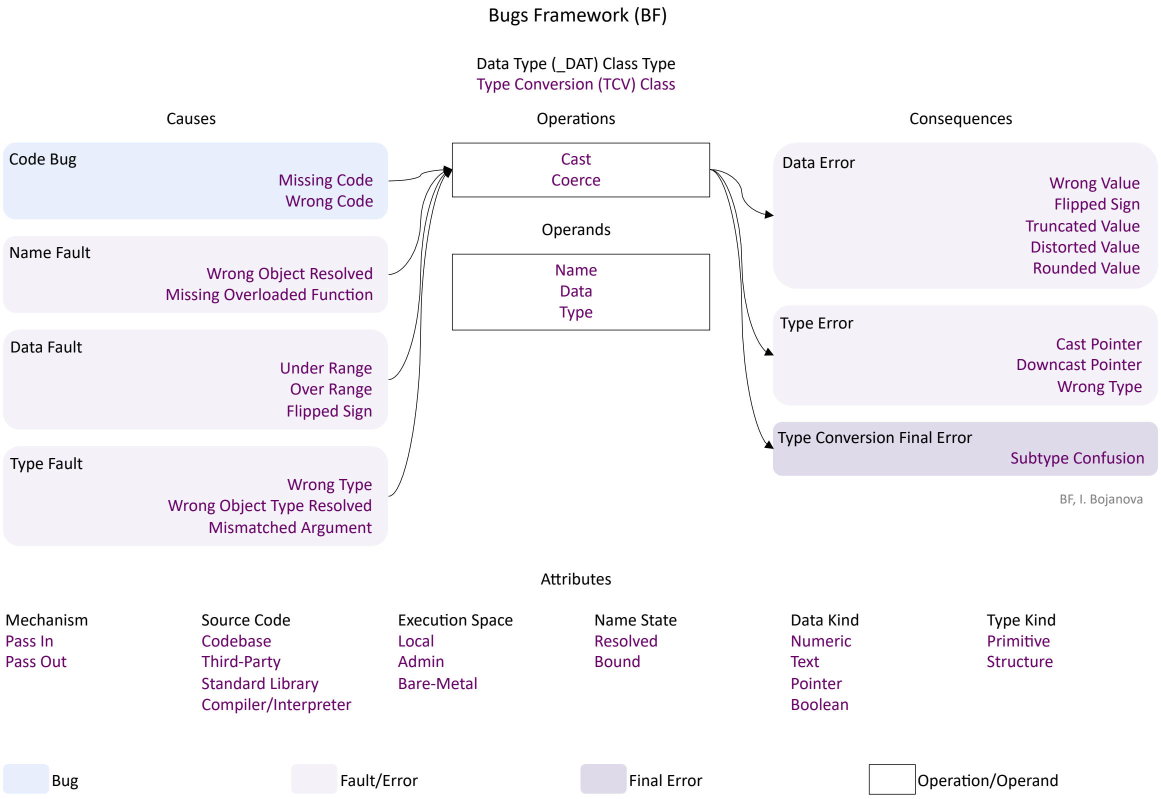


**Fig. 13.** BF Type Conversion (TCV) class

Each BF class taxonomy also defines a matrix of semantic rules for propagation between weaknesses, as *error* to *fault* match is always by type but may be on different levels of abstraction by value (for details, see the forthcoming SPs 800-231Cx and [1]).

Formally, the specification of a security weakness is an instance of a BF weakness class with one cause, one operation, one consequence, and operation and operand attribute values from the sets with value taxons of that class. The *operation* binds the causation within a weakness as a ⟨*cause*, *operation*⟩→*consequence* relation that must adhere to the within-weakness causation semantic rules for that class.

For example, the two most severe weaknesses — *missing validation of input data leads to a SQL query injection* and *writing data via a pointer beyond the upper bound of an array leads to a buffer overflow* [27] — are specified with BF as follows: ⟨*Missing Code*, *Validate*⟩→ *Query Injection* and ⟨*Overbound Pointer*, *Write*⟩→*Buffer Overflow*.

The BF class taxonomies with built-in taxon definitions are available in machine-readable formats (e.g., see the BF in XML format in Fig. 14 and query it via the BF API at [1]).

The type and value taxon definitions are visualized in the graphical representation of the BF class taxonomies (e.g., see the BF MUS class at [1]) and BFCVE specifications (e.g., the BF CVE-2014-0160 specification at [1]), and as tooltips of the BF tool (see Sec. 10.3).

**Table 2.** Type taxon definition examples

| Name | Definition |
|---|---|
| Specification Bug | A defect in the metadata or algorithm of an operation (i.e., proper operands over an improper operation). It is the root cause of a security vulnerability. |
| Data Fault | The data of an object has harmed semantics or an inconsistent or wrong value. |
| Injection | An exploitable or undefined system behavior caused by validation or sanitization bugs. |
| Memory Corruption/ Disclosure | An exploitable or undefined system behavior caused by memory addressing, allocation, use, or deallocation bugs. |

**Table 3.** Value taxon definition examples

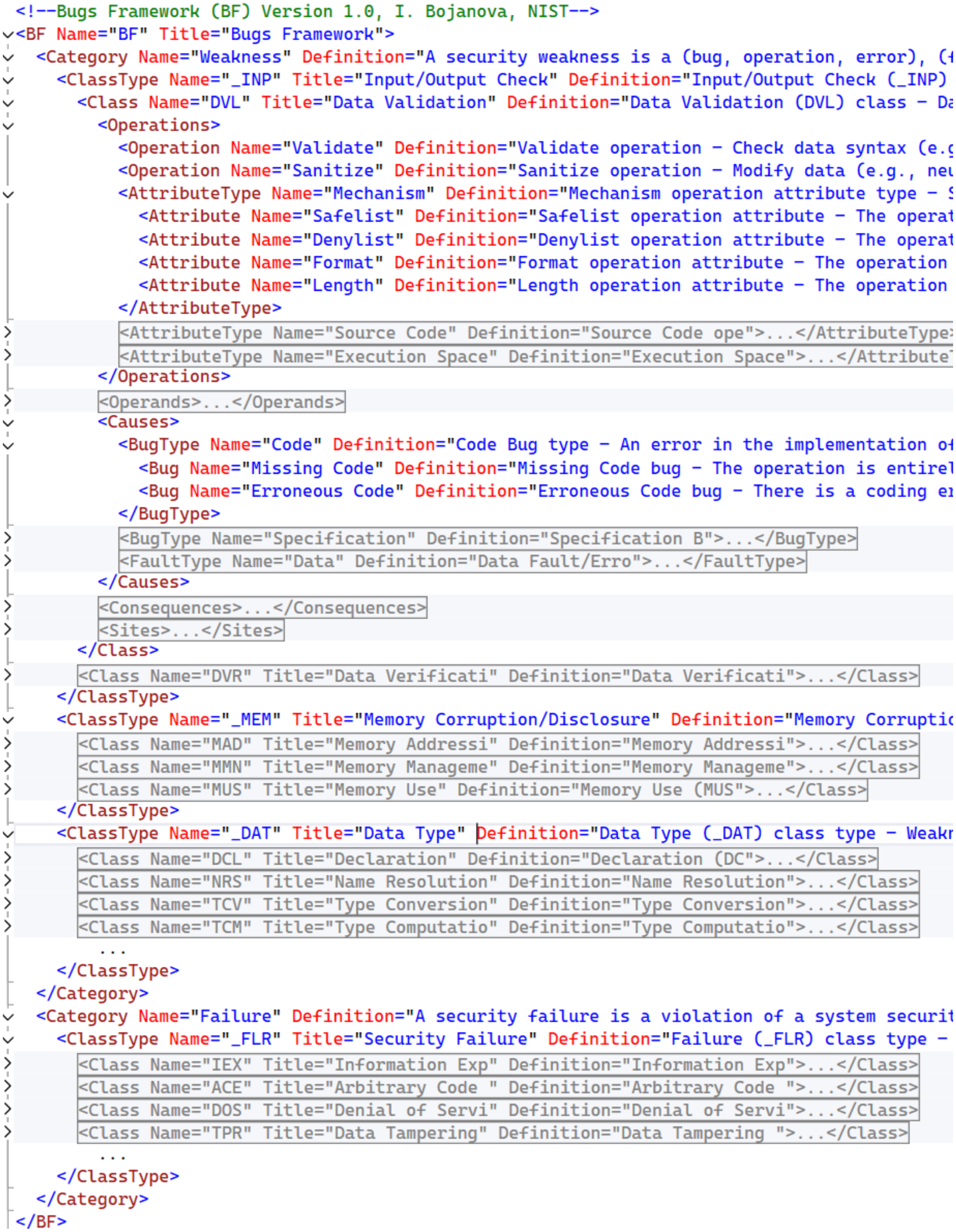
| Name | Definition |
|---|---|
| Under-Restrictive Policy | Accepts bad data. For example, permissive safe list or regular expression, or incomplete deny list. |
| Wrong Size | The value used as size or length (i.e., the number of elements) does not match the object's memory size or length. |
| Sanitize | Modify data (e.g., neutralize/escape, filter/remove, repair/add symbols) to make it valid (well-formed). |
| Write | Change the data value of an object in memory to another meaningful value. |
| Query Injection | Maliciously inserted condition parts (e.g., `or 1 == 1`) or entire commands (e.g., `drop table`) into an input used to construct a query (e.g., SQL or NoSQL Injection, XPath Injection, XQuery Injection, or LDAP Injection). |
| Buffer Overflow | Write data above the upper bound of an object (i.e., buffer overwrite). |

**Table 4.** Operation attribute types

| Name | Definition |
|---|---|
| Mechanism | Shows how the operation with a bug or faulty operand is performed. |
| Source Code | Shows where the code of the operation with a bug or faulty operand resides. |
| Execution Space | Shows where the operation with a bug or faulty operand is executed and the privilege level at which it runs. |

**Table 5.** Operand attribute types

| Name | Definition |
|---|---|
| Name Kind | Shows what the entity with this name is. |
| Name State | Shows what the stage of the entity name is. |
| Data Kind | Shows what the type or category of data is. |
| Data State | Shows where the data comes from. |
| Type Kind | Shows what the data type composition is. |
| Address Kind | Shows how much memory is accessed (i.e., the span) outside of a bound of an object. |
| Address State | Shows where the address is (i.e., its location) in the memory layout. |
| Size Kind | Shows what is used as the size or length (i.e., the number of elements) of an object. |



```
<!--Bugs Framework (BF) Version 1.0, I. Bojanova, NIST-->
<BF Name="BF" Title="Bugs Framework">
  <Category Name="Weakness" Definition="A security weakness is a (bug, operation, error), (
    <ClassType Name="_INP" Title="Input/Output Check" Definition="Input/Output Check (_INP)
      <Class Name="DVL" Title="Data Validation" Definition="Data Validation (DVL) class – Da
        <Operations>
          <Operation Name="Validate" Definition="Validate operation – Check data syntax (e.g
          <Operation Name="Sanitize" Definition="Sanitize operation – Modify data (e.g., neu
          <AttributeType Name="Mechanism" Definition="Mechanism operation attribute type – S
            <Attribute Name="Safelist" Definition="Safelist operation attribute – The operat
            <Attribute Name="Denylist" Definition="Denylist operation attribute – The operat
            <Attribute Name="Format" Definition="Format operation attribute – The operation
            <Attribute Name="Length" Definition="Length operation attribute – The operation
          </AttributeType>
          <AttributeType Name="Source Code" Definition="Source Code ope">...</AttributeType
          <AttributeType Name="Execution Space" Definition="Execution Space">...</Attribute
        </Operations>
        <Operands>...</Operands>
        <Causes>
          <BugType Name="Code" Definition="Code Bug type – An error in the implementation of
            <Bug Name="Missing Code" Definition="Missing Code bug – The operation is entirel
            <Bug Name="Erroneous Code" Definition="Erroneous Code bug – There is a coding er
          </BugType>
          <BugType Name="Specification" Definition="Specification B">...</BugType>
          <FaultType Name="Data" Definition="Data Fault/Erro">...</FaultType>
        </Causes>
        <Consequences>...</Consequences>
        <Sites>...</Sites>
      </Class>
      <Class Name="DVR" Title="Data Verificati" Definition="Data Verificati">...</Class>
    </ClassType>
    <ClassType Name="_MEM" Title="Memory Corruption/Disclosure" Definition="Memory Corruptio
      <Class Name="MAD" Title="Memory Addressi" Definition="Memory Addressi">...</Class>
      <Class Name="MMN" Title="Memory Manageme" Definition="Memory Manageme">...</Class>
      <Class Name="MUS" Title="Memory Use" Definition="Memory Use (MUS">...</Class>
    </ClassType>
    <ClassType Name="_DAT" Title="Data Type" Definition="Data Type (_DAT) class type – Weakn
      <Class Name="DCL" Title="Declaration" Definition="Declaration (DC">...</Class>
      <Class Name="NRS" Title="Name Resolution" Definition="Name Resolution">...</Class>
      <Class Name="TCV" Title="Type Conversion" Definition="Type Conversion">...</Class>
      <Class Name="TCM" Title="Type Computatio" Definition="Type Computatio">...</Class>
        ...
    </ClassType>
  </Category>
  <Category Name="Failure" Definition="A security failure is a violation of a system securit
    <ClassType Name="_FLR" Title="Security Failure" Definition="Failure (_FLR) class type –
      <Class Name="IEX" Title="Information Exp" Definition="Information Exp">...</Class>
      <Class Name="ACE" Title="Arbitrary Code " Definition="Arbitrary Code ">...</Class>
      <Class Name="DOS" Title="Denial of Servi" Definition="Denial of Servi">...</Class>
      <Class Name="TPR" Title="Data Tampering" Definition="Data Tampering ">...</Class>
        ...
    </ClassType>
  </Category>
</BF>
```

**Fig. 14.** BF taxonomy in XML

## 6.2. BF Failure Class

A BF failure class defines sets of possible exploit vectors used for the exploits of a specific vulnerability to result in the loss of security properties. The exploit vectors propagate from the final errors of BF weakness classes.

The BF *Failure (_FLR)* class type encompasses strictly defined failure classes, such as:

- *Information Exposure (IEX)* — Inadvertent disclosure of information that leads to confidentiality loss.
- *Arbitrary Code Execution (ACE)* — Execution of unauthorized commands or code execution that could lead to everything being lost.

  Remote code execution (RCE) is a sub-case of ACE on a target system or device from a remote location, typically over a network.
- *Denial of Service (DOS)* — Disruption of access to or use of information or information systems that leads to availability loss.
- *Data Tampering (TPR)* — Unauthorized modification or destruction of information that leads to integrity loss.

An IEX, ACE, or TPR failure may result in a fault that starts a new chained vulnerability.

## 6.3. BF Methodology

The methodology for developing BF bugs models and weakness classes involves the following 12 steps (also see Fig. 15):

1. Phases: Analyse common weakness types (including CWEs) and publicly disclosed vulnerabilities (including CVEs) and identify related software, firmware, or hardware execution phases in which specific types of bugs could be introduced and faults propagated. Each execution phase would be the basis for defining a new BF *weakness class*. The BF classes of related execution phases would define a new BF *class type*.

   For example, the MAD and MUS classes (see Fig. 12) correspond to the related memory addressing and memory use execution phases. They are also of the BF _MEM class type [1, 17].
2. Operations and Operands: Identify the operations and their input operands for each execution phase so that all BF classes remain orthogonal by operation. They would define the possible values of *operations* for the $\langle$*cause*, *operation*$\rangle \rightarrow$*consequence* weakness triples for each of the new BF classes.

   For example, the DVL class has two operations (see Fig. 11), MUS has four operations (see Fig. 12), TCV has two operations (see Fig. 13), and their sets of operations do not overlap.

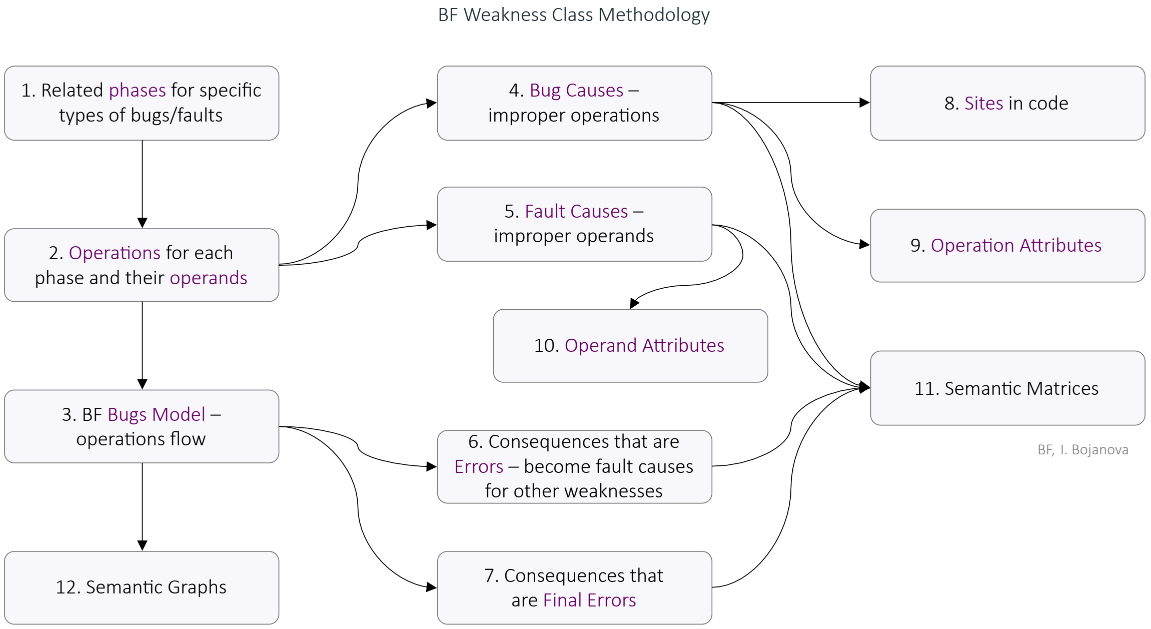

BF Weakness Class Methodology
1. Related phases for specific types of bugs/faults
2. Operations for each phase and their operands
3. BF Bugs Model – operations flow
12. Semantic Graphs
4. Bug Causes – improper operations
5. Fault Causes – improper operands
10. Operand Attributes
6. Consequences that are Errors – become fault causes for other weaknesses
7. Consequences that are Final Errors
8. Sites in code
9. Operation Attributes
11. Semantic Matrices
BF, I. Bojanova

**Fig. 15.** BF class methodology

3. Bugs Model: Define the BF bugs model reflecting the identified phases, operations, and the proper flow between the operations. It would be the basis for the definition of a *semantic graph* of meaningful operation flow.

   For example, the BF Memory Bugs Model covers the MAD, MAL, MUS, and MDL phases and the flow between their operations (see Fig. 9).

4. Bug Causes: Identify the possible code and specification defects for the operations of each phase. They would define the *bug* values of the ⟨*bug*, *operation*⟩→*error* or ⟨*bug*, *operation*⟩→*final error* weakness triples (see Fig. 3) for each of the new BF classes.

   For example, the DVL class has two values for each of the *Code Bug* and *Code Specification* bug types — *Missing Code* and *Erroneous Code*, and *Under-Restrictive Policy* and *Over-Restrictive Policy*, respectively (see Fig. 11).

5. Fault Causes: Identify which of the *Name*, *Data*, *Type*, *Address*, and *Size* input operands apply to the operations of each phase. They would define the possible *fault* and *error types*. Identify the possible operand errors, which would define the *fault* values of the ⟨*fault*, *operation*⟩→*error* or ⟨*fault*, *operation*⟩→*final error* weakness triples (see Fig. 3) for each of the new BF classes.

   For example, the MUS class has *Data Fault*, *Type Fault*, *Address Fault*, *Size Fault*, and *Data Error* types — the first four as causes, the last one as a consequence. It also has 11 fault values (see Fig. 12).

6. Error Consequences: Identify the possible output result errors from the operations that propagate as faults of other weaknesses (i.e., improper input operands for other operations). These would define the *error* values of the weakness triples (see Fig. 3).

   For example, there is one error value for DVL — *Invalid Data* (see Fig. 11), one for MUS — *Uninialized Object* (See Fig. 12), and eight for TCV — five of *Data Error* and three of *Type Error* type (see Fig. 13).

7. Final Error Consequences: Identify the possible output result errors from the operations that do not propagate to faults of other weaknesses and instead propagate to exploit vectors toward failures. They would define the *final error* values of the weakness triples (see Fig. 3).

   For example, MUS has one final error type — *Memory Corruption/Disclosure* — with eight final error values (see Fig. 12).

8. Sites: Identify syntactic places in code where such bugs or faults can occur. This step is mainly applicable to low-level bugs and faults.

9. Operation Attributes: Identify specific descriptive values for the *Execution Space*, *Mechanism*, and Source Code operation attribute types for each of the new BF classes.

   For example, DVL has the *Safelist*, *Denylist*, *Format*, and *Length* values for the Mechanism attribute type (see Fig. 11), while TCV has *Pass In* and *Pass Out* (see Fig. 13).

10. Operand Attributes: Identify specific descriptive values for the relevant *Name*, *Data*, *Type*, *Address*, and *Size* attribute types for each of the new BF classes.

    For example, the TCV class has the *Resolved* and *Bound* values for the *Name State* operand attribute type; *Numeric*, *Text*, *Pointer*, and *Boolean* values for *Data Kind*; and *Primitive* and *Structure* values for *Type Kind* (see Fig. 13).

11. Semantic Matrices: Identify the meaningful ⟨*cause*, *operation*⟩→*consequence* causal relations for each of the new BF classes, and same-type different-value *consequence*↷*cause* propagations for classes of different BF class types.

12. Semantic Graph: Define the graph of meaningful operation flow based on the BF Bugs Model.

Finally, create the BF weakness class taxonomies in machine-readable formats, generate graphical representations to enhance understanding, regenerate the BF LL(1) CFG to include the new taxonomies, and update the BFDB database (see Sec. 10).

The methodology for developing BF failure classes is analogous but also simpler, as final errors of BF weakness classes directly match to exploit vectors of BF Failure classes, and the *exploit*, at least for now, is on an abstract level.

For more BF classes and details, refer to the forthcoming SPs 800-231Cx, *Bugs Framework: _yyy Taxonomy*, where _yyy is a BF class type.

# 7. BF Vulnerability Models

The BF *vulnerability models* define state and specification views of a security vulnerability as a chain of weaknesses linked by causality that may converge and chain with other vulnerabilities to enable harmful failures. The state view represents the weaknesses as improper-state (*operation*, *operand*$_1$, $\ldots$, *operand*$_n$) tuples and their causal transitions. The specification view reflects the BF taxonomic representation of a weakness as a $\langle$*cause*, *operation*$\rangle\rightarrow$*consequence* relation and *consequence*$\curvearrowright$*cause* between weaknesses propagation.

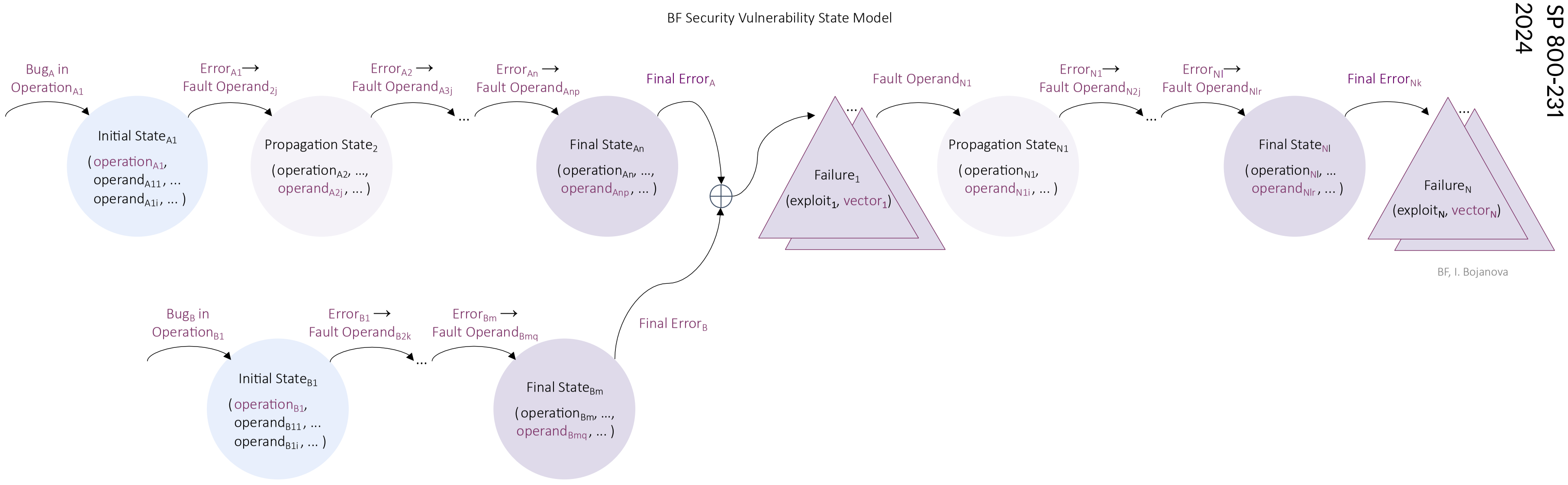
## 7.1. BF Vulnerability State Model

The BF Vulnerability State Model defines a *vulnerability* as deterministic state automata of improper states and their transitions (see Fig. 16). A transition is to another weakness or to a failure. An improper state is an (*operation*, *operand*$_1$, $\ldots$, *operand*$_n$) tuple with at least one improper element (depicted in purple). A BF operation is the minimal input-process-output code that — because of a bug or fault — results in an error that propagates to another fault or is final (see Sec. 3.1). A transition is defined by the erroneous result from the operation over the input operands (i.e., the output of the improper state).

The *initial state* corresponds to a weakness caused by a bug in the operation or a hardware defect-induced fault of an operand. It results in an error or — if it is the only state — is a final error (i.e., an undefined or exploitable system behavior). A *propagation state* corresponds to a weakness that is caused by a fault of an operand and results in an error. The *final state* corresponds to a weakness caused by a fault of an operand or — if it is the only state — by a bug in the operation. It results in a final error that introduces an exploit vector that enables a failure. The initial state relates to the root cause of the vulnerability, and the final state relates to its sink. For simplicity, Fig. 16 does not detail vulnerability chains that start with a hardware defect-induced bug or fault, as Fig. 4 does.

Fixing the bug or a hardware defect-induced fault will resolve the vulnerability, while fixing a non-hardware-defect-induced fault will only mitigate it. Fixing a bug may relate to fixing a design flaw, such as an unaccounted-for system configuration or environment. For a one-chain improper states example, recall the BadAlloc pattern in Fig. 5.

Vulnerabilities composition is via convergence at their final errors or failure-to-fault-based chaining. In some cases, for an exploit at the sink to be harmful, several vulnerabilities must converge (depicted with $\oplus$ in Fig. 16) at their final errors. Fixing the root cause (i.e., the bug) of at least one of the chains would avoid the failure. An exploit of a vulnerability may also result in a failure that creates a fault that starts a new vulnerability of only fault-type weaknesses. There must be an exploit for the failure to occur and a fault that results from it to start the new chain (see the gaps between the arrows and failures in Fig. 16 that indicate that there is no direct weakness-to-weakness transition there). Fixing the root cause (i.e., the bug) of the first vulnerability will resolve the entire chain of vulnerabilities.


BF Security Vulnerability State Model
Bug$_A$ in Operation$_{A1}$
Initial State$_{A1}$ (operation$_{A1}$, operand$_{A11}$, … operand$_{A1i}$, … )
Error$_{A1}$ → Fault Operand$_{2j}$
Propagation State$_2$ (operation$_{A2}$, …, operand$_{A2j}$, … )
Error$_{A2}$ → Fault Operand$_{A3j}$
…
Error$_{An}$ → Fault Operand$_{Anp}$
Final State$_{An}$ (operation$_{An}$, …, operand$_{Anp}$, … )
Final Error$_A$
Failure$_1$ (exploit$_1$, vector$_1$)
…
Fault Operand$_{N1}$
Propagation State$_{N1}$ (operation$_{N1}$, operand$_{N1i}$, … )
Error$_{N1}$ → Fault Operand$_{N2j}$
…
Error$_{Nl}$ → Fault Operand$_{Nlr}$
Final State$_{Nl}$ (operation$_{Nl}$, … operand$_{Nlr}$, … )
Final Error$_{Nk}$
Failure$_N$ (exploit$_N$, vector$_N$)
…
BF, I. Bojanova
Bug$_B$ in Operation$_{B1}$
Initial State$_{B1}$ (operation$_{B1}$, operand$_{B11}$, … operand$_{B1i}$, … )
Error$_{B1}$ → Fault Operand$_{B2k}$
…
Error$_{Bm}$ → Fault Operand$_{Bmq}$
Final State$_{Bm}$ (operation$_{Bm}$, …, operand$_{Bmq}$, … )
Final Error$_B$

Improper State – An operation, operand$_1$, …, operand$_n$) tuple with at least one improper element (depicted in purple)

● Initial State – a weakness caused by a Bug in the Operation, resulting in an Error

● Propagation State – a weakness caused by a Fault of an Operand, resulting in an Error

● Final State – a weakness caused by a Fault in an Operand, resultingin a Final Error

△ Failure – a violation of a security requirement caused by an exploit; may result in a Fault starting a new faults only vulnerability

↷ Chaining weaknesses underlying a vulnerability

⊕ Converging vulnerabilities

**Fig. 16.** BF Vulnerability State model

For example, in Fig. 16, *Chain A* starts from the *Initial State*$_{A1}$, where *operation*$_{A1}$ has a software, firmware, or hardware defect-induced bug. The *Error*$_{A1}$ result propagates to a fault of *operand*$_{A2j}$, which leads to the *Propagation State*$_{A2}$. The last *operation*$_{An}$ in this chain with faulty *operand*$_{Anp}$ results in the *Final Error*$_{A}$. *Chain B* analogously propagates through improper states, and its *Final Error*$_{B}$ converges with *Final Error*$_{A}$ toward *Failure*$_{1}$ and possibly more failures. Once the exploit vector introduced by the final errors is used to exploit the vulnerability, *Failure*$_{1}$ possibly creates a faulty *operand*$_{AN1}$ that starts a new vulnerability chain, and so on until the final security *Failure*$_{N}$ is reached.

Heartbleed, CVE-2014-0160 — as a real-world example — was a severe vulnerability in the OpenSSL cryptographic library [23]. A server (or client) with a vulnerable heartbeat extension would *bleed* data via a small heartbeat message with a large requested *length* (i.e., larger than the actual array *size*). Each exploit could reveal up to 64KB of raw memory of highly sensitive information (e.g., private keys and login credentials) via buffer over-reads. However, NVD labels it with CWE-125: Out-of-bounds Read, which covers both under-lower-bound and over-upper-bound reads from a buffer. In addition, it reflects only the weakness with the final error at the sink, not the weakness with the bug as the root cause.

The BF state view of Heartbleed is presented in Fig. 17 as two converging vulnerability chains of underlying weaknesses. The BF taxonomy helps identify and comprehensively label three weaknesses in the main chain and one more in the secondary chain.

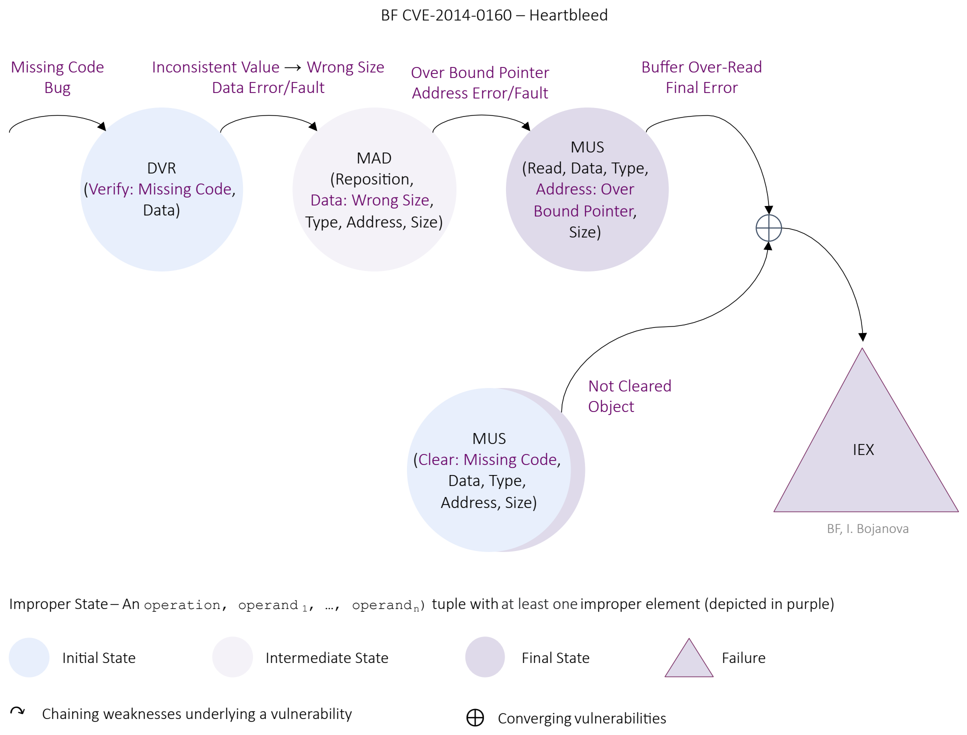


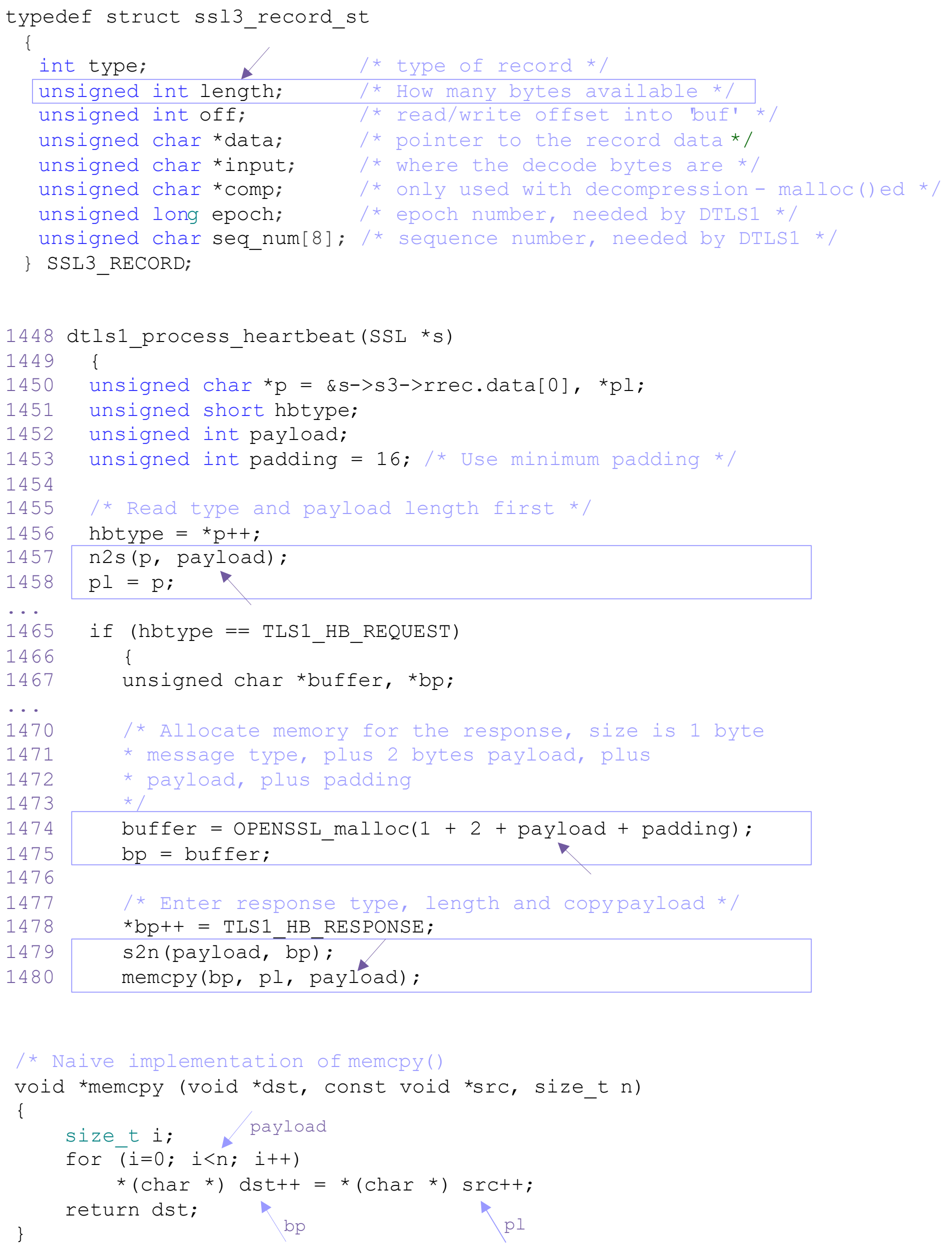
**Fig. 17.** BF states of Heartbleed

The bug was both in the ssl\d1_both.c and ssl\t1_lib.c files of the OpenSSL implementation of the TLS protocol [28]. Analysis of the C code before the fix (see Fig. 18 for d1_both.c) shows that the very first improper BF state is in the data verification phase, where the semantics of the input should be checked and corrected. The pointer *p* (see line 1450 in Fig. 18) is to a record of type SSL3_RECORD (see the top of Fig. 18) whose second field holds the length. The payload variable is declared as an unsigned int (see line 1452) and can be a huge number. It is assigned the value of the length field of p via the n2s macro (see line 1457). That is input data that supplies the length of an array (i.e., a buffer), but it is not checked before use toward the actual array size (i.e., the number of elements in the record data). Its value is not verified. This BF bug state is the first of a chain of improper states that would lead to buffer over-read. It is an instance of the BF Data Verification (DVR) class [1] as the (*Verify: Missing Code*, *Data*) tuple with an improper *Verify* operation element (see the first state in Fig. 17) — the entire data verification operation is absent — that results in an *Inconsistent Value* error.

Then, memcpy() reads payload number of bytes from the object pointed by pl and copies them to the object pointed by bp (see line 1480 in Fig. 18). Following the naive C implementation of memcpy() at the bottom in Fig. 18, bp and pl are passed by reference via the dst and src arguments, and the huge payload length is passed via the n argument. First, one byte is read from pl and copied to bp. Until the huge payload length is reached, both pointers move one byte up, and the newly pointed by pl byte is read and copied. However, while bp is allocated large enough at up to 1+2+65535+16 bytes (see lines 1474 and 1475 in Fig. 18), pl points to an array with a reasonable size (see line 1458). As the content of this array is read and copied to bp, so too is a huge amount of data from over its upper bound.

The analysis reveals two fault states: when pl is repositioned over the array upper bound and when data values are read from there. The former is an instance of the BF Memory Addressing (MAD) class [1] as the (*Reposition*, *Data: Wrong Size*, *Type*, *Address*, *Size*) tuple with an improper *Data* operand element (see the second state in Fig. 17) that results in an *Overbound Pointer* error. There is no bug in the *Reposition* operation itself, but a value that is inconsistent with the size of the pl object is used to control the iteration. The latter is an instance of the BF Memory Use (MUS) class [1] as the (*Read*, *Data*, *Type*, *Address: Overbound Pointer*, *Size*) tuple with an improper *Address* operand element (see the third state in Fig. 17) that results in a *Buffer Over-Read* final error. Again, there is no bug in the *Read* operation itself, but because pl points overbound, it is possible to read data that should not be read (i.e., buffer over-read).

The three-state BF chain so far (see the upper row in Fig. 17) shows that data can be read from over the bound of the array pointed by pl. However, it does not show why an exploit would reach sensitive information, such as private keys or login credentials. The vulnerability triggered by the missing size verification bug is only a part (although the main one) of the puzzle.



```
typedef struct ssl3_record_st
 {
  int type;                    /* type of record */
  unsigned int length;         /* How many bytes available */
  unsigned int off;            /* read/write offset into 'buf' */
  unsigned char *data;         /* pointer to the record data */
  unsigned char *input;        /* where the decode bytes are */
  unsigned char *comp;         /* only used with decompression - malloc()ed */
  unsigned long epoch;         /* epoch number, needed by DTLS1 */
  unsigned char seq_num[8];    /* sequence number, needed by DTLS1 */
 } SSL3_RECORD;
```

```
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 copypayload */
     *bp++ = TLS1_HB_RESPONSE;
     s2n(payload, bp);
     memcpy(bp, pl, payload);
```

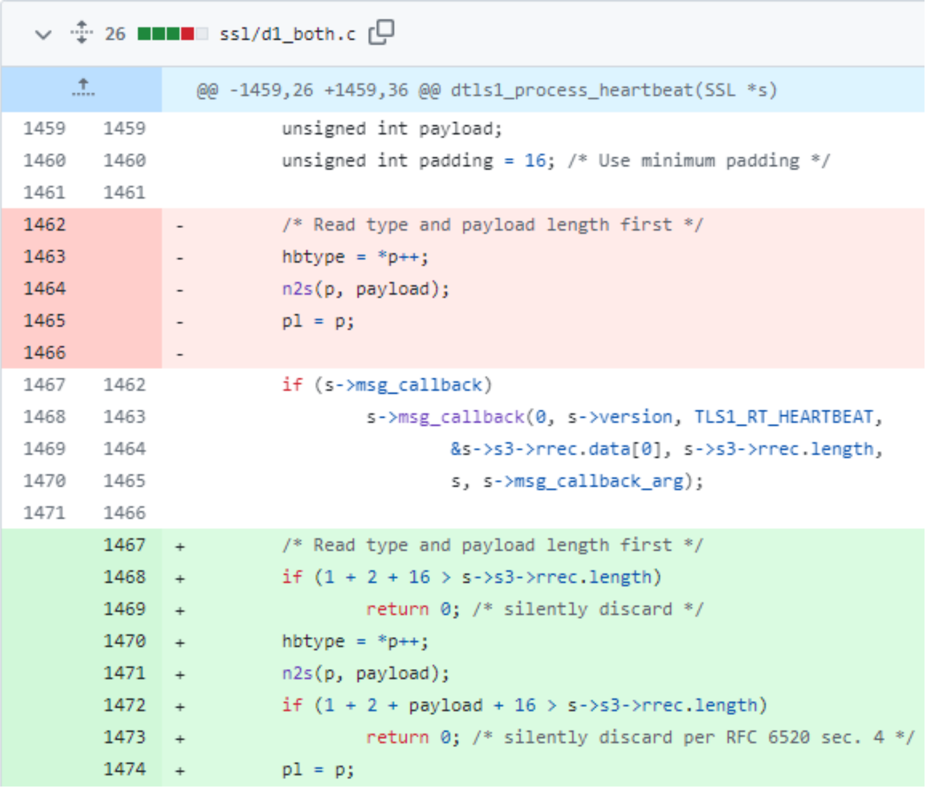
```
/* Naive implementation of memcpy()
void *memcpy (void *dst, const void *src, size_t n)
{
    size_t i;
    for (i=0; i<n; i++)
        *(char *) dst++ = *(char *) src++;
    return dst;
}
```

**Fig. 18.** C code of `heartbeat()` and naive `memcpy()`

There must have been another coding error due to which an unused object with sensitive data was left in memory unaware of the risks. The bug state of this parallel vulnerability is again an instance of the BF MUS class but as the (*Clear: Missing Code*, *Data*, *Type*, *Address*) tuple with an improper *Clear* operation (see the second chain in Fig. 17) that results in a *Not Cleared Object* final error. Converging the final errors from both chains (i.e., buffer over-read and not cleared object), the vulnerable software can now reach and expose sensitive information.

The bug and fault state automata of Heartbleed (see Fig. 17) expresses it as two converging vulnerability chains of underlying weaknesses. Missing input data verification leads to the use of inconsistent size for a buffer and allows for a pointer reposition over its bound, which — converging with missing clear — allows for remote reads and the exposure of sensitive information. Multiple exploits of Heartbleed, each exposing up to 64KB of memory, can accumulate huge amounts of data, such as "secret keys used for certificates, user names and passwords, instant messages, emails, and business-critical documents and communication" [29].

The fix of the bug in the main Heartbleed chain was to add input data semantics checks and silently ignore the heartbeat message if the requested length was larger than the actual array size (see Fig. 19) [30]. Lines 1468 and 1469 discard heartbeats with zero length. Lines 1472 and 1473 ensure that the actual length of the record data is sufficiently large.



```
26 ssl/d1_both.c

            @@ -1459,26 +1459,36 @@ dtls1_process_heartbeat(SSL *s)
1459  1459          unsigned int payload;
1460  1460          unsigned int padding = 16; /* Use minimum padding */
1461  1461
1462        -       /* Read type and payload length first */
1463        -       hbtype = *p++;
1464        -       n2s(p, payload);
1465        -       pl = p;
1466        -
1467  1462          if (s->msg_callback)
1468  1463                  s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1469  1464                          &s->s3->rrec.data[0], s->s3->rrec.length,
1470  1465                          s, s->msg_callback_arg);
1471  1466
      1467  +       /* Read type and payload length first */
      1468  +       if (1 + 2 + 16 > s->s3->rrec.length)
      1469  +               return 0; /* silently discard */
      1470  +       hbtype = *p++;
      1471  +       n2s(p, payload);
      1472  +       if (1 + 2 + payload + 16 > s->s3->rrec.length)
      1473  +               return 0; /* silently discard per RFC 6520 sec. 4 */
      1474  +       pl = p;
```

**Fig. 19.** Heartbleed fix in Heartbeat

For more details on the BF Vulnerability State Model, refer to the forthcoming SP 800-231D, *Bugs Framework: Vulnerability Models*.

### 7.2. BF Vulnerability Specification Model

The BF Vulnerability Specification Model defines a *vulnerability specification* as a chain of ⟨*cause*, *operation*⟩→*consequence* relations (i.e., weakness triples) with operation and operand attributes and *consequence*↷*cause* between weaknesses propagation (see Fig. 20). The model reflects the BF taxonomy structure (see Sec. 6) and the BF Vulnerability State Model (see Sec. 7.1). For simplicity, Fig. 20 does not visualize vulnerability convergence and chaining as Fig. 16 does. However, see Fig. 21 for a demonstrative example of vulnerability convergence.

The BF allows for the expression of a weakness as a ⟨*cause*, *operation*⟩→*consequence* causal triple with operation and operand attributes. A cause is a bug in an operation or a fault of an operand, and a consequence is the erroneous result from the operation. Bugs are code or specification defects, and faults are input operand defects. The output errors from operations propagate to faults or are final errors that introduce exploit vectors toward failures. A fault is of a *name*, *data*, *type*, *address*, or *size*. (see Sec. 3 and 4).

Causation within a weakness is by meaningful ⟨*cause*, *operation*⟩→*consequence* relations. That is, the sets of valid relations defined for each BF taxonomy (see Sec. 6) restrict it. The bug or faulty input operand of an operation results in an error or a final error as a valid ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, or ⟨*fault*, *operation*⟩→*final error* weakness triple. For example, ⟨*Under-Restrictive Policy*, *Validate*⟩→*Source Code Injection* and ⟨*Mismatched Argument*, *Coerce*⟩→*Truncated Value* are meaningful weakness triples, but ⟨*Underbound Pointer*, *Write*⟩→*Buffer Overflow* is not, as the pointer is below the lower bound while the *Write* is over the upper bound.

Causation between weaknesses is by *error type* to *fault type* match, and *error value*↷*fault value* match or — for weaknesses of different BF class types — a meaningful values propagation (see Sec. 6). It is also guided by the valid flow of operations defined by the BF Bugs Models. If the causation between weaknesses does not follow the proper operation flow, an operation must be missing, which indicates an *Missing Code* bug. For example, ⟨*Wrong Type*, *Coerce*⟩→*Flipped Sign* ↷⟨*Wrong Argument*, *Evaluate*⟩→*Under Range* is a valid weakness causation because the triples specify valid within-weakness relations, the *Evaluate* operation may follow the *Coerce* operation (see Fig. 10), and *Flipped Sign*↷*Wrong Argument* is a valid data error-to-fault by value propagation.

The ⟨*Erroneous Code*, *Verify*⟩→*Inconsistent Value*↷⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer*↷⟨*Overbound Pointer*, *Write*⟩→*Buffer Overflow* are valid weakness causations, as the data and address by type, *Inconsistent Value*↷*Wrong Size* by value, and *Overbound Pointer* exact value propagation are all valid. For a similar one-chain example, recall the BadAlloc pattern from Sec. 3.3, Fig. 5 and see the BF CVE-2021-21834 specification at [1]). For a convergence involving example, see the BF specification of Heartbleed below.

Causation between vulnerabilities is by *exploit result type*↷*fault type* propagation (i.e., the fault starts a new faults-only vulnerability). For example, exposed private keys may become the fault that starts a new vulnerability.

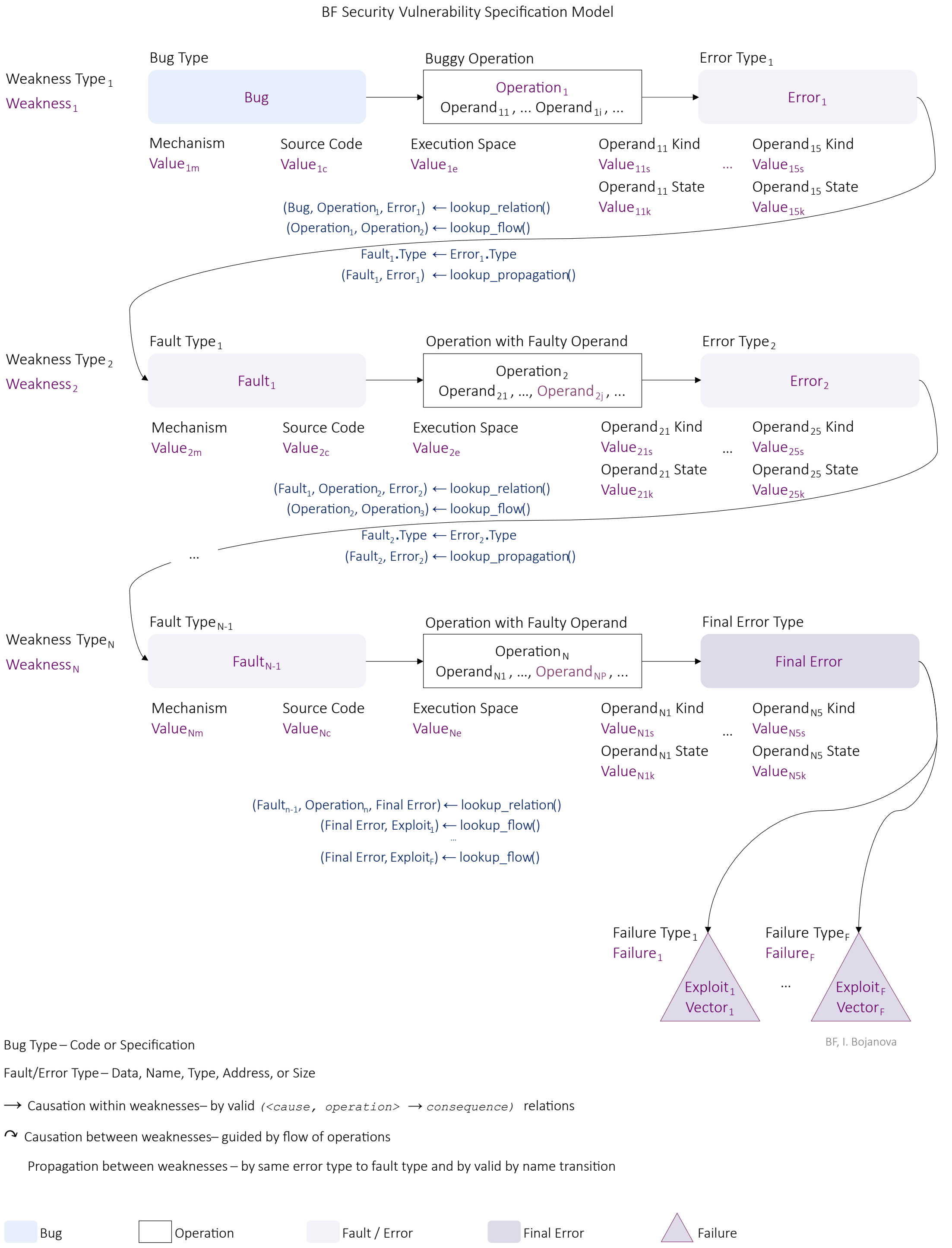

BF Security Vulnerability Specification Model
Weakness $\text{Type}_1$
$\text{Weakness}_1$
Bug Type
Bug
Buggy Operation
$\text{Operation}_1$
$\text{Operand}_{11}$, ... $\text{Operand}_{1i}$, ...
Error $\text{Type}_1$
$\text{Error}_1$
Mechanism
$\text{Value}_{1m}$
Source Code
$\text{Value}_{1c}$
Execution Space
$\text{Value}_{1e}$
$\text{Operand}_{11}$ Kind
$\text{Value}_{11s}$
$\text{Operand}_{11}$ State
$\text{Value}_{11k}$
...
$\text{Operand}_{15}$ Kind
$\text{Value}_{15s}$
$\text{Operand}_{15}$ State
$\text{Value}_{15k}$
(Bug, $\text{Operation}_1$, $\text{Error}_1$) ⟵ lookup_relation()
($\text{Operation}_1$, $\text{Operation}_2$) ⟵ lookup_flow()
$\text{Fault}_1$.Type ⟵ $\text{Error}_1$.Type
($\text{Fault}_1$, $\text{Error}_1$) ⟵ lookup_propagation()
Weakness $\text{Type}_2$
$\text{Weakness}_2$
Fault $\text{Type}_1$
$\text{Fault}_1$
Operation with Faulty Operand
$\text{Operation}_2$
$\text{Operand}_{21}$, ..., $\text{Operand}_{2j}$, ...
Error $\text{Type}_2$
$\text{Error}_2$
Mechanism
$\text{Value}_{2m}$
Source Code
$\text{Value}_{2c}$
Execution Space
$\text{Value}_{2e}$
$\text{Operand}_{21}$ Kind
$\text{Value}_{21s}$
$\text{Operand}_{21}$ State
$\text{Value}_{21k}$
...
$\text{Operand}_{25}$ Kind
$\text{Value}_{25s}$
$\text{Operand}_{25}$ State
$\text{Value}_{25k}$
($\text{Fault}_1$, $\text{Operation}_2$, $\text{Error}_2$) ⟵ lookup_relation()
($\text{Operation}_2$, $\text{Operation}_3$) ⟵ lookup_flow()
$\text{Fault}_2$.Type ⟵ $\text{Error}_2$.Type
($\text{Fault}_2$, $\text{Error}_2$) ⟵ lookup_propagation()
...
Weakness $\text{Type}_N$
$\text{Weakness}_N$
Fault $\text{Type}_{N-1}$
$\text{Fault}_{N-1}$
Operation with Faulty Operand
$\text{Operation}_N$
$\text{Operand}_{N1}$, ..., $\text{Operand}_{NP}$, ...
Final Error Type
Final Error
Mechanism
$\text{Value}_{Nm}$
Source Code
$\text{Value}_{Nc}$
Execution Space
$\text{Value}_{Ne}$
$\text{Operand}_{N1}$ Kind
$\text{Value}_{N1s}$
$\text{Operand}_{N1}$ State
$\text{Value}_{N1k}$
...
$\text{Operand}_{N5}$ Kind
$\text{Value}_{N5s}$
$\text{Operand}_{N5}$ State
$\text{Value}_{N5k}$
($\text{Fault}_{n-1}$, $\text{Operation}_n$, Final Error) ⟵ lookup_relation()
(Final Error, $\text{Exploit}_1$) ⟵ lookup_flow()
...
(Final Error, $\text{Exploit}_F$) ⟵ lookup_flow()
Failure $\text{Type}_1$
$\text{Failure}_1$
$\text{Exploit}_1$
$\text{Vector}_1$
...
Failure $\text{Type}_F$
$\text{Failure}_F$
$\text{Exploit}_F$
$\text{Vector}_F$
BF, I. Bojanova
Bug Type – Code or Specification
Fault/Error Type – Data, Name, Type, Address, or Size
⟶ Causation within weaknesses – by valid (<cause, operation> ⟶ consequence) relations
↷ Causation between weaknesses – guided by flow of operations
Propagation between weaknesses – by same error type to fault type and by valid by name transition
Bug
Operation
Fault / Error
Final Error
Failure

**Fig. 20.** BF Vulnerability Specification Model

For example, the BF specification view of Heartbleed, CVE-2014-0160, is presented in Fig. 21. It expands the Heartbleed improper states view (see Fig. 17) via the BF taxonomic representation of a weakness as a ⟨*cause*, *operation*⟩→*consequence* triple with attributes and its *consequence*↷*cause* propagation.

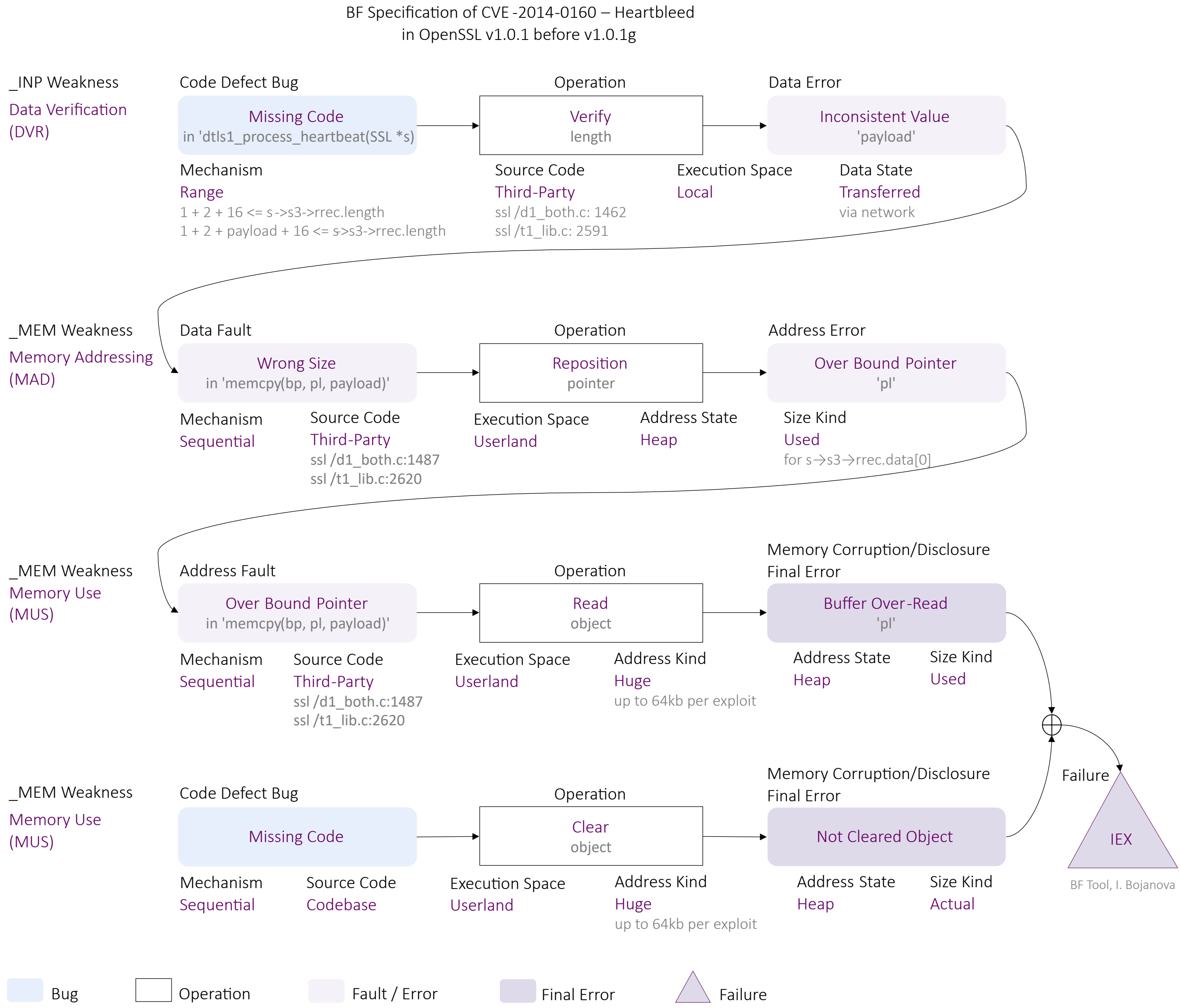


**Fig. 21.** BF specification of Heartbleed

Using the BF taxonomies of the involved weakness types, the ⟨*Missing Code*, *Verify*⟩→ *Inconsistent Value* weakness (see the first triple in the first chain in Fig. 21) is an instance of the BF DVR class. The missing input data verification (i.e., semantics check) security bug leads to a *Data Error* — a data value that is inconsistent with the size of the array.

The operation and operand attributes provide details on *what*, *how*, and *where* it went wrong. The *Mechanism*, *Source Code*, and *Execution Space* attributes are about the *Verify* operation. *Mechanism* shows that the missing verification should have been checked against range (i.e., the actual length). *Source Code* shows that the bug is in third-party software — the `d1_both.c` and `t1_lib.c` files. *Execution Space* shows that the code with the bug is running in an environment with local user (i.e., limited) permissions. The *Data State* attribute is about the *Data* operand and shows that the data was transferred.

Next, the ⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer* weakness (see the second triple in the first chain in Fig. 21) is an instance of the BF MAD class. The wrong size *Data Fault* at repositioning leads to a pointer pointing overbound (i.e., an address error). The *Mechanism* attribute for this weakness shows that the repositioning is sequential and iterates over the buffer elements. The *Execution Space* is userland — an environment with privilege levels but in unprivileged mode. The *Address State* attribute shows that the buffer is dynamically allocated in the heap. The *Size Kind* attribute shows that the iteration over the elements of the buffer is limited by a used value (supplied with the request); it is not limited by the actual size of the array.

Last in this chain, the ⟨*Overbound Pointer*, *Read*⟩→*Buffer Over-Read* weakness (see the third triple in the first chain in Fig. 21) is an instance of the BF MUS class. The overbound pointer *Address Fault* results in a buffer over-read *Memory Disclosure Final Error*. The *Address Kind* attribute shows that the accessed out-of-bounds memory is huge — up to 64KB of memory per request.

The converging vulnerability (see the second chain in Fig. 21) chain comprises another BF MUS instance — a ⟨*Missing Code*, *Clear*⟩→*Not Cleared Object* weakness. The missing clear (change to a non-meaningful value, such as via zeroization) bug leads to an object with not cleared data — a memory disclosure final error. The attributes are the same as for the BF MUS weakness in the main vulnerability (see Chain 1 in Fig. 21). However, this is a different vulnerability, and the source code is in different software.

Combined, the memory disclosure final errors *Buffer Over-Read* and *Not Cleared Object* cause an information exposure (IEX) security failure. Either the missing *Verify* bug or the missing *Clear* bug has to be fixed to avoid this security failure.

The corresponding BF specification of Heartbleed — CVE-2014-0160.bfcve — in XML format is shown in Fig. 26. For more details, see BF CVE-2014-0160 Heartbleed at [1].

For more details on the BF Vulnerability Specification Model, refer to the forthcoming SP 800-231D, *Bugs Framework: Vulnerability Models*.

## 8. BF Formal Language

The BF formal language is generated by the BF left-to-right leftmost derivation one-symbol lookahead (LL(1)) attribute context-free grammar (ACFG) derived from the BF CFG. Since it is based on an LL(1) grammar, the BF formal language is guaranteed to be unambiguous, and the BF weakness and vulnerability specifications are guaranteed to be clear and precise.[2] The BF lexis, syntax, and semantics are based on the BF structured causal taxonomies (e.g., Fig. 12), bugs models (e.g., Fig. 9), and vulnerability models (e.g., Fig. 16 and 20). *Lexis* refers to the vocabulary (i.e., words and symbols) used by a specification language. *Syntax* is about validating the grammatical structure (i.e., the form) of a specification. *Semantics* is about verifying the logical structure (i.e., the meaning) of a specification.

The *BF CFG* is a powerful tool for specifying and analyzing security weaknesses and vulnerabilities. It is defined in Listing 1 as a four-tuple

$$G = (V, \Sigma, R, S), \tag{1}$$

where:

- $\Sigma$ defines the BF lexis (i.e., the alphabet of the CFG) as a finite set of tokens (terminals) comprised by the sets of BF taxons and BF symbols

  $$\Sigma = \big\{\alpha \mid \alpha \in \Sigma Taxon \cup \Sigma Symbol$$

- $V$ and $R$ define the BF syntax (i.e., the types of phrases and the rules of the CFG) as
  - A finite set of variables (nonterminals)

    $$V = \big\{S, V_1, \ldots, V_n$$

    and
  - A finite set of syntactic rules (productions) in the form

    $$R = \big\{A \longmapsto \omega \mid A \in V \wedge \omega \in \big(V \cup \Sigma\big)^* \quad ,$$

    where:

    $\big(V \cup \Sigma\big)^*$ is a string of tokens and/or variables, and

    $A \longmapsto \omega$ means that any variable $A$ occurrence may be replaced by $\omega$.
- $S \in V$ is the predefined start variable from which all BF specifications derive.

[2] *Clear* means easy to understand, straightforward, and unambiguous with no room for confusion or misinterpretation. *Precise* means exact, accurate, and specific, which also implies unambiguous.

A BF *specification* starts from $S$ and ends with the empty string. The *derivation* is via a sequence of steps in which nonterminals are replaced by the right-hand side of a production. The production rules are applied to a variable regardless of its context.

The *BF formal language* is generated by the BF LL(1) ACFG $G = (V, \Sigma, R, S)$ (see Listing 8) that augments the syntax of the BF CFG with semantic rules (see Listing 7). It is defined in Listing 2 as the set $L(G)$ of all strings of tokens $\omega$ derivable from the start variable $S$.

$$L(G) = \{\omega \in \Sigma^* : S \overset{*}{\Longrightarrow} \omega\}, \tag{2}$$

where:

- $\Sigma^*$ is the set of all possible strings that can be generated from $\Sigma$ tokens
- $S$ is the start variable
- $\alpha \overset{*}{\Longrightarrow} \beta$ means string $\alpha$ derives string $\beta$.

Strings involving nonterminals are not part of the language (i.e., $\omega$ must be in $\Sigma^*$ — the set of strings made from terminals).

### 8.1. BF Lexis

The BF formal language lexis refers to the vocabulary of the BF formal language: the set of tokens $\Sigma$. Listing 3 defines it as the set of BF taxons (see Sec. 6) and the set of symbols for converging and chaining vulnerabilities (see Fig. 16):

$$\Sigma = \left\{ \Sigma Taxon, \Sigma Symbol \right. \tag{3}$$

$$\begin{aligned} \Sigma Taxon = \Big\{ & Operation,\ BugType,\ Bug,\ FaultType,\ Fault,\ ErrorType,\ Error, \\ & FinalErrorType,\ FinalError, OperationAttribute,\ OperandAttribute,\ \ldots \end{aligned}$$

$$\begin{aligned} \Sigma Symbol = \Big\{ & CausationSymbol, ChainingSymbol, ConvergingSymbol, \\ & SemicolonSymbol, CommaSymbol, LeftAngleSymbol, RightAngleSymbol \end{aligned}$$

The BF CFG lexis defines the BF taxons (e.g., for BF operations, bugs, faults, final errors, operation attributes, operand attributes, and failures). It also defines the set of BF symbols for specifying causation within a weakness ($\rightarrow$), chaining weaknesses or vulnerabilities ($\curvearrowright$), and converging vulnerabilities ($\oplus$). The taxons are in quotes (e.g., *'Missing Code'* or *'Query Injection'*) and considered literal words.

The BF classes are of a *'Weakness'* or *'Failure'* category. Listing 4 provides an excerpt of the BF lexis expressed via the Extended Backus–Naur Form (EBNF) [31] using the following meta-notations:

| Symbol | Meaning |
|---|---|
| $=$ | defining |
| $\mid$ | definition separator |
| ; | terminator |

$$Category = 'Weakness' \mid 'Failure'; \tag{4}$$

$$ClassType = '\_INP' \mid '\_MEM' \mid '\_DAT' \mid \ldots$$

$$Class = 'DVL' \mid 'DVR' \mid 'MAD' \mid 'MMN' \mid 'MUS' \mid 'DCL' \mid 'NRS' \mid 'TCV' \mid 'TCM' \mid \ldots;$$

$$\begin{aligned} Operation \; = \; & 'Validate' \mid 'Sanitize' \mid 'Verify' \mid 'Correct' \mid 'Initialize\ Pointer' \mid 'Dereference' \\ \mid \; & 'Reposition' \mid 'Reassign' \mid 'Allocate' \mid 'Extend' \mid 'Reallocate-Extend' \\ \mid \; & 'Reallocate-Reduce' \mid 'Reduce' \mid 'Deallocate' \mid 'Initialize\ Object' \mid 'Read' \\ \mid \; & 'Write' \mid 'Clear' \mid 'Declare' \mid 'Define' \mid 'Refer' \mid 'Call' \mid 'Cast' \mid 'Coerce' \\ \mid \; & 'Calculate' \mid 'Evaluate' \; \ldots; \end{aligned}$$

$$BugType = 'Code\ Bug' \mid 'Specification\ Bug';$$

$$\begin{aligned} Bug \; = \; & 'Missing\ Code' \mid 'Erroneous\ Code' \mid 'Mismatched\ Operation' \mid \ldots \\ \mid \; & 'Under-Restrictive\ Policy' \mid 'Over-Restrictive\ Policy' \\ \mid \; & 'Missing\ Modifier' \mid 'Wrong\ Modifier' \\ \mid \; & 'AnonymousScope' \mid 'WrongScope' \\ \mid \; & 'Missing\ Qualifier' \mid 'Wrong\ Qualifier' \mid \ldots; \end{aligned}$$

$$FaultType = 'Name\ Fault' \mid 'Data\ Fault' \mid 'Type\ Fault' \mid 'Address\ Fault';$$

$$\begin{aligned} Fault \; = \; & 'Missing\ Overridden\ Function' \mid 'Missing\ Overloaded\ Function' \\ \mid \; & 'Wrong\ Object\ Resolved' \mid 'Wrong\ Function\ Resolved' \mid \ldots; \\ \mid \; & 'Corrupted\ Data' \mid 'Tampered\ Data' \mid 'Corrupted\ Policy\ Data' \\ \mid \; & 'Tampered\ Policy\ Data' \mid 'Invalid\ Data' \mid 'NULL\ Pointer' \\ \mid \; & 'Hardcoded\ Address' \mid 'Single\ Owned\ Address' \mid 'Wrong\ Index' \mid 'Wrong\ Size' \\ \mid \; & 'Flipped\ Sign' \mid 'Wrong\ Argument' \mid 'Reference\ vs.\ Dereference' \mid \ldots \\ \mid \; & 'Cast\ Pointer' \mid 'Wrong\ Type' \mid 'Wrong\ Index\ Type' \mid 'Insufficient\ Size' \\ \mid \; & 'Downcast\ Pointer' \mid 'Wrong\ Argument\ Type' \mid 'Wrong\ Object\ Type\ Resolved' \mid \ldots \\ \mid \; & 'Wild\ Pointer' \mid 'Dangling\ Pointer' \mid 'Untrusted\ Pointer' \\ \mid \; & 'Overbound\ Pointer' \mid 'Underbound\ Pointer' \mid 'Wrong\ Position\ Pointer' \mid \ldots \end{aligned}$$

$$ErrorType = 'Name\ Error' \mid 'Data\ Error' \mid 'Type\ Error' \mid 'Address\ Error';$$

$$\begin{aligned} FinalErrorType = {} & 'Injection\ FinalError' \mid 'Memory\ Corruption/Disclosure\ FinalError' \mid \\ & 'Entity\ Access\ FinalError' \end{aligned}$$

$$\begin{aligned} FinalError\ = {} & 'Query\ Injection' \mid 'Command\ Injection' \mid 'Source\ Code\ Injection' \mid \ldots; \\ & \mid 'NULL\ Pointer\ Dereference' \mid 'Untrusted\ Pointer\ Dereference' \\ & \mid 'Uninitialized\ Pointer\ Dereference' \mid 'Memory\ Leak' \mid 'Memory\ Overflow' \\ & \mid 'Double\ Deallocate'\ \mid 'Object\ Corruption' \mid 'Not\ Cleared\ Object' \\ & \mid 'Type\ Confusion' \mid 'Use\ After\ Deallocate' \mid 'Buffer\ Overflow' \\ & \mid 'Buffer\ Underflow' \mid 'Buffer\ Over-Read' \mid 'Buffer\ Under-Read' \\ & \mid 'Subtype\ Confusion' \mid 'Undefined' \mid \ldots; \\ \ldots & \end{aligned}$$

$$\begin{aligned} CausationSymbol &= '\rightarrow'; & SemicolonSymbol &= ':'; & LeftAngleSymbol &= '\langle'; \\ ChainingSymbol &= '\curvearrowright'; & CommaSymbol &= ','; & RightAngleSymbol &= '\rangle' \\ ConvergingSymbol &= '\oplus';; \end{aligned}$$

## 8.2. BF Syntax

The BF formal language syntax is about validating the grammatical structure of a BF specification. It adheres to the BF production rules (i.e., nonterminals) for constructing (producing) valid specifications of the language that correspond to the BF Vulnerability Specification Model structure and flow (see Fig. 20), including the converging and chaining rules (see Fig. 16) defined by the BF Vulnerability State Model.

The BF CFG syntax defines a vulnerability that possibly converges with other vulnerabilities, leading to one or more failures. The CFG production rules are expressed via the EBNF using the following meta-notations:

| Symbol(s) | Meaning |
|---|---|
| $=$ | defining |
| $\mid$ | definition separator |
| $[\ ]$ | option — zero or one occurrences |
| $\{\ \}$ | repetition — zero or more occurrences |
| $(\ )$ | grouping |
| ; | terminator |

(5)

$$
\begin{aligned}
S &= Vulnerability,\ '\curvearrowright',\ Failure;\\
Vulnerability &= WeaknessChain,\ \{'\oplus',\ WeaknessChain\};\\
WeaknessChain &= Weakness,\ \{'\curvearrowright',\ Weakness\};\\
Weakness &= '\langle',\ Cause,\ ',',\ Operation,\ '\rangle',\ '\rightarrow',\ Consequence;\\
Cause &= Bug\\
&\mid Fault;\\
Consequence &= Error\\
&\mid FinalError;\\
Error\ &= Fault;
\end{aligned}
$$

A vulnerability is defined as a chain of weaknesses, possibly converged and chained with other vulnerabilities. A weakness is defined as a ⟨*cause, operation*⟩→ *consequence* triple. A cause is defined as a bug or a fault. A consequence is defined as an error or a final error (see Listing 5). However, according to the BF Vulnerability Specification Model (see Fig. 20), only the cause of the first weakness can be a bug, and only the last consequence can be a final error.

The last production in Listing 5 expresses that the same set of taxons corresponds to *Fault* and *Error*, although they are different non-terminals — the former is a cause of a weakness, the latter is a consequence of a weakness. An *Error* $\curvearrowright$ *Fault* propagation may be on a different level of abstraction (e.g., *Inconsistent Value* $\curvearrowright$ *Wrong Size*).

A vulnerability with a single weakness is the only case in which a weakness is defined with both a bug cause and a final error consequence. A propagation weakness is caused by a fault and results in an error. Listing 6 reflects these rules in the productions of Listing 5 and eliminates the *Cause* and *Consequence* variables.

(6)

$$
\begin{aligned}
S &= Vulnerability,\ '\curvearrowright',\ Failure;\\
Vulnerability &= WeaknessChain,\ \{'\oplus',\ WeaknessChain\};\\
WeaknessChain &= SingleWeakness\\
&\mid FirstWeakness,\ \{'\curvearrowright',\ Weakness\},\ '\curvearrowright',\ LastWeakness;\\
SingleWeakness &= '\langle',\ (Bug \mid Fault),\ ',',\ Operation,\ '\rangle',\ '\rightarrow',\ FinalError;\\
FirstWeakness &= '\langle',\ Bug \mid Fault),\ ',',\ Operation,\ '\rangle',\ '\rightarrow',\ Error;\\
Weakness &= '\langle',\ Fault,\ ',',\ Operation,\ '\rangle',\ '\rightarrow',\ Error;\\
LastWeakness &= '\langle',\ Fault,\ ',',\ Operation,\ '\rangle',\ '\rightarrow',\ FinalError;
\end{aligned}
$$

To ensure that the BF specifications are unambiguous, the next step is to demonstrate the successful derivation of a BF LL(1)[3] formal grammar from the BF CFG. A CFG is an *LL(1)*

[3] The most restrictive LL(1) is chosen for the simplicity and efficiency of parser implementations.

*grammar* if and only if only one token (terminal) or variable (nonterminal) is needed to make a parsing decision [32]. LL(1) grammars are *not ambiguous* and *not left-recursive*.

The *BF LL(1) formal CFG* is derived from the BF EBNF productions on Listing 6 via left factorization and left recursion elimination. It is suitable for recursive descent parsing, as the start of each production option is unique. The rule to choose on each step is uniquely determined by the current variable and the next taxon (if there is one).

Listing 7 defines the BF LL(1) CFG production rules for constructing valid, unambiguous BF specifications. Compared to Listing 6, it also details the bug, fault, error, and final error type non-terminals.

(7)

$$
\begin{aligned}
S &= \mathit{Vulnerability}, \mathit{Converge\_Failure};\\
\mathit{Vulnerability} &= {}'\langle', \mathit{Bug\_Fault}, ',', \mathit{Operation}, '\rangle', '\rightarrow',\\
&\quad \mathit{OperAttrs\_Error\_FError};\\
\mathit{Bug\_Fault} &= \mathit{BugType}, ':', \mathit{Bug}\\
&\mid \mathit{T\_Fault};\\
\mathit{OperAttrs\_Error\_FError} &= \mathit{OperationAttribute}, \mathit{OperAttrs\_Error\_FError}\\
&\mid \mathit{ErrorType}, ':', \mathit{Error}, '\curvearrowright',\\
&\quad '\langle', \mathit{T\_Fault}, ',', \mathit{OprndAttrs\_Operation}\\
&\mid \mathit{FinalErrorType}, ':', \mathit{FinalError};\\
\mathit{OprndAttrs\_Operation} &= \mathit{OperandAttribute}, \mathit{OprndAttrs\_Operation}\\
&\mid \mathit{Operation}, '\rangle', '\rightarrow', \mathit{OperAttrs\_Error\_FError};\\
\mathit{Converge\_Failure} &= {}'\oplus', \mathit{Vulnerability}, \mathit{Converge\_Failure}\\
&\mid \ \rightarrow', \mathit{Failure};\\
\mathit{T\_Fault} &= \mathit{FaultType}, ':', \mathit{Fault};
\end{aligned}
$$

The BF specifications are derived from the start symbol $S$ by step-by-step production application, substituting for the leftmost terminal one at a time until the string is fully expanded (i.e., consists of only terminals).

## 8.3. BF Semantics

The BF formal language semantics is about verifying the logical structure of a BF specification. It is defined by extending the BF LL(1) CFG to a BF LL(1) ACFG with static semantic rules that adhere to the BF Vulnerability Models causation and propagation rules (see Fig. 16 and 20). The static semantic rules are expressed via a set of grammar attributes (i.e., properties to which values can be assigned), a set of semantic functions for computing the attribute values, and a possibly empty set of predicate functions for each production rule (e.g., Donald Knuth's attribute grammars [33]).

Listing 8 presents the BF LL(1) ACFG syntax and semantic rules. If a nonterminal appears in more than one rule, it gets subscripted. The semantic rules prevent invalid within weaknesses relations and *error*$\curvearrowright$*fault* by *value* between weaknesses propagation and check for valid flow by operation.

The BF LL(1) ACFG adds the *Type* synthesized attribute for the nonterminals *Fault* and *Error* to store the operand types (i.e., *Name*, *Data*, *Type*, *Address*, or *Size*) and *FinalError* to store the final error types (e.g., *Injection*, *Memory Corruption/Disclosure*, *Access*, and *Type Compute*). The predicates express propagation by *error type* and *fault type*.

(8)

Syntax Rules:

$$\begin{aligned}
S \;&=\; Vulnerability,\; Converge\_Failure;\\
Vulnerability \;&=\; '\langle',\; Bug\_Fault,\; ',',\; Operation_1,\; '\rangle',\; '\rightarrow',\\
&\phantom{=}\;\; OperAttrs\_Error\_FError;\\
Bug\_Fault \;&=\; BugType,\; ':',\; Bug\\
&\;\;\mid\; FaultType,\; ':',\; Fault;\\
OperAttrs\_Error\_FError \;&=\; OperationAttribute,\; OperAttrs\_Error\_FError\\
&\;\;\mid\; ErrorType,\; ':',\; Error,\; '\curvearrowright',\\
&\phantom{=}\;\; '\langle',\; FaultType,\; ':',\; Fault_1,\; ',',\; OprndAttrs\_Operation\\
&\;\;\mid\; FinalErrorType,\; ':',\; FinalError;\\
OprndAttrs\_Operation \;&=\; OperandAttribute,\; OprndAttrs\_Operation\\
&\;\;\mid\; Operation_k,\; '\rangle',\; '\rightarrow',\; OperAttrs\_Error\_FError;\\
Converge\_Failure \;&=' \oplus',\; Vulnerability,\; Converge\_Failure\\
&\;\;\mid\; Failure;\\
T\_Fault \;&=\; FaultType,\; ':',\; Fault;
\end{aligned}$$

Semantic Rules:

$$\begin{aligned}
\big(Bug,\; Operation_1,\; Error\big) &\leftarrow lookup\_relation()\\
\big(Bug,\; Operation_1,\; FinalError\big) &\leftarrow lookup\_relation()\\
\big(Fault_1,\; Operation_k,\; Error\big),\; k>1 &\leftarrow lookup\_relation()\\
\big(Fault_1,\; Operation_k,\; FinalError\big),\; k>1 &\leftarrow lookup\_relation()\\
(Operation_1,\; \ldots\;,\; Operation_k),\; k>1 &\leftarrow lookup\_flow()\\
Fault_1 \;\leftarrow\; if\; (Fault_1.ClassType \;==\; Error.ClassType)&\; then\, Error\\
else\; (Fault_1,\; Error) &\leftarrow lookup\_propagation()
\end{aligned}$$

Predicates:

$$\begin{aligned}
Fault_1.Type \;&==\; Error.Type\\
ExploitVector.Type \;&==\; FinalError.Type
\end{aligned}$$

For example, listing 9 expresses the formal BF specification of CVE-2014-0160 Heartbleed.

$$
\begin{gathered}
BF\_INP\ DVR \\
\langle Code\ Bug:\ Missing\ Code,\ Verify\rangle \rightarrow\ Data\ Error:\ Inconsistent\ Value \\
Mechanism:\ Range,\ Source\ Code:\ Third-Party,\ Execution\ Space:\ Local \\
Data\ State:\ Transferred \\
\curvearrowright \\
\\
BF\_MEM\ MAD \\
\langle Data\ Fault:\ Wrong\ Size,\ Reposition\rangle \rightarrow\ Address\ Error:\ Over\ Bound\ Pointer \\
Mechanism:\ Sequential,\ Source\ Code:\ Third-Party,\ Execution\ Space:\ Userland \\
Address\ State:\ Heap,\ Size\ Kind:\ Used \\
\curvearrowright \\
\\
BF\_MEM\ MUS \\
\langle Address\ Fault:\ Over\ Bound\ Pointer,\ Read\rangle \rightarrow Memory\ Corruption/Disclosure \\
Final\ Error:\ Buffer\ Over-Read \\
Mechanism:\ Sequential,\ Source\ Code:\ Third-Party,\ Execution\ Space:\ Userland \\
Size\ Kind:\ Used,\ Address\ Kind:\ Huge,\ Address\ State:\ Heap \\
\oplus \\
\\
BF\_MEM\ MUS \\
\langle Code\ Bug:\ Missing\ Code,\ Clear\rangle \rightarrow Memory\ Corruption/Disclosure \\
Final\ Error:\ Not\ Cleared\ Object \\
Mechanism:\ Sequential,\ Source\ Code:\ Third-Party,\ Execution\ Space:\ Userland \\
Address\ Kind:\ Huge,\ Address\ State:\ Heap,\ Size\ Kind:\ Actual \\
\curvearrowright \\
\\
IEX
\end{gathered}
\tag{9}
$$

For the fully expressed BF lexis, syntax, and semantics in EBNF, refer to the forthcoming SP 800-231E, *Bugs Framework: Formal Language*.

## 9. BF Secure Coding Principles

The Software Engineering Institute (SEI) Computer Emergency Response Team (CERT) Coding Standards (e.g., [34] and [35]) and the Open Worldwide Application Security Project (OWASP) Secure Coding Practices [36] set the current state of the art in secure coding. They provide rules and practices that are grouped by topic and described in natural language. The CERT rules are also programming language-specific, though they do provide useful non-compliant code examples and compliant solutions.

In contrast, the BF bugs models, weakness and failure taxonomies, and vulnerability models (see Sec. 5, 6, and 7) form the basis for the formal definition of secure coding principles by software, firmware, or hardware execution phases, that are also programming language-independent. For example, the BF Input/Output Check Bugs Model (see Fig. 8) and classes (see BF _INP at [1]) address input/output check safety (e.g., no SQL injections or use of wrong input values). The BF Memory Bugs Model (see Fig. 9) and classes (see BF _MEM at [1]) address memory safety (e.g., no use after frees or buffer overflows). The BF Data Type Bugs Model (see Fig. 10) and classes (see BF _DAT at [1]) address data type safety (e.g., no integer overflows or subtype confusions). The BF Vulnerability Models rules (see Sec. 7), which are reflected in the BF semantics, help identify the dependencies between different kinds of code safety.

The BF bugs models define the sets of operations where code safety could break. They also define the proper operation flow within and between execution phases that — if not followed — could also break code safety. The x-axis of a model reflects temporal safety. The y-axis of a model may reflect spatial safety. The BF weakness taxonomies are organized by the bugs models phases and define why (i.e., bugs and faults), where (i.e., operations), and how (i.e., errors and final errors) the code safety could break. The BF vulnerability models define the causation, propagation, and convergence rules, which in addition to the proper operation flow, define how breaking one kind of code safety may lead to breaking another.

### 9.1. Input/Output Check Safety

*Input/output check safety* ensures the use of proper input/output data in code. That is, data is properly validated and sanitized and/or verified and corrected. It is addressed by the BF Input/Output Check Bugs Model (see Fig. 8) operation flow and the BF DVL and DVR classes (see BF _INP at [1]) that define why, where, and how input/output check safety could break. It relates to the BF data operations *Validate*, *Sanitize*, *Verify*, and *Correct*. Input/output data must be validated (syntax check) and then sanitized (escaped, filtered, or repaired) and/or verified (semantics check) and then corrected (assigned a new value or removed), if needed.

Avoiding the meaningful $\langle$*bug/fault*, *operation*$\rangle$ couples of the BF _INP classes would guarantee input/output check safety. That is, avoiding the BF DVL bugs (i.e., missing or erroneous validation, or under-restrictive or over-restrictive validation policy) and faults (i.e., corrupted or tampered data or validation policy) guarantees safety from errors, such as *Invalid Data* and final errors, such as *Query Injection* (e.g., SQL injection) and *Source Code Injection* (e.g., Cross Site Scripting (XSS)). Injections enable the following security failures: IEX, TPR, ACE, and its sub-case RCE. Avoiding the BF DVR bugs (i.e., missing or erroneous verification or under-restrictive or over-restrictive verification) and faults (i.e., invalid data) guarantees safety from errors, such as *Wrong Value*, *Inconsistent Value*, and *Wrong Type*.

## 9.2. Memory Safety

*Memory safety* ensures the proper access and use of memory in code. That is, pointers to objects are properly initialized, dereferenced, repositioned, or reassigned, and objects are properly allocated, initialized, read, written, resized, cleared, or deallocated. It is addressed by the BF Memory Bugs Model (see Fig. 9) operation flow and the BF MAD, MMN, and MUS classes (see BF _MEM at [1]) that define why, where, and how memory safety could break.

Memory safety has both temporal and spatial aspects that depend on pointer safety. *Temporal memory safety* ensures that an object memory is only accessed or used during its life cycle and only via its proper pointers (owners). Access is via BF MAD *Dereference* of a pointer to the object; use is via BF MUS *Read* or *Write* of object data. The first operation over an allocated object must be BF MUS *Initialize Object*, and the last one before it is deallocated must be BF MUS *Clear* (see Fig. 9).

Examples of temporal memory safety are uninitialized object, use after deallocate (i.e., use after free or use after return in C), and double deallocate (i.e., double free in C) safety. The first prevents the use of non-meaningful data values, the second prevents the use of data values via dangling pointers, and the third prevents the deallocation of deallocated objects via dangling pointers. The following BF weakness specifications detail what bugs or faults could break these three kinds of temporal memory safety: ⟨*Missing Code/Erroneous Code*, *Initialize Object*⟩→*Uninitialized Object*, ⟨*Dangling Pointer*, *Read*⟩→*Use After Deallocate*, and ⟨*Dangling Pointer*, *Deallocate*⟩→*Double Deallocate*. A dangling pointer holds the address of its successfully deallocated object (i.e., a pointer to a freed heap object or address of a stack object returned by a function) and is the consequence of a ⟨*Missing Code*, *Reassign*⟩→*Dangling Pointer* (see the discussion on pointer safety below) after a *Deallocate* operation.

*Spatial memory safety* ensures access or use within the bounds of an allocated object and only via its pointers (owners). In addition to the MAD *Dereference* and MUS *Read* and *Write* operations, it also relates to the MAD and MDL operations along the y-axis of the _MEM Bugs Model (see Fig. 9) that affect the object boundaries: *Allocate*, *Extend*, *Reallocate-Extend*, *Reduce*, and *Reallocate-Reduce*. The size of the object is always strictly defined, and the pointer must not exceed its boundaries.

Examples of spatial memory safety are buffer overflow and underflow safety, and buffer over-read and under-read safety. The following BF weakness specifications detail what bugs or faults could break these two kinds of spatial memory safety: ⟨*Overbound Pointer*, *Write*⟩→*Buffer Overflow* and ⟨*Underbound Pointer*, *Read*⟩→*Buffer Under-Read*. For consideration, there are also the fault weaknesses that may cause them, such as ⟨*Wrong Size*, *Allocate/Reduce*⟩→ *Insufficient Size*, ⟨*Wrong Size*, *Reposition*⟩→*Underbound Pointer*, and ⟨*Insufficient Size*, *Reposition*⟩→*Overbound Pointer*, which in turn are caused by the declaration of verification bug weaknesses. Buffer overflows and underflows enable TPR, DOS, and ACE failures. Buffer over-reads and under-reads enable IEX failures.

Allocation in excess or failure to deallocate unused objects (see the MMN *Memory Overflow* and *Memory Leak* final errors, correspondingly) could exhaust memory. The former impacts spatial memory safety. The latter directly impacts temporal safety and indirectly impacts spatial memory safety. Both enable DOS failures.

*Pointer Safety* ensures that an object is only accessed via its proper pointers (owners). It relates to the MAD, MAL, and MDL operations (see Fig. 9) that assign or reassign the object pointer (owner): *Initialize Pointer*, *Allocate*, and *Reassign*. Use of *Wild Pointer*, *Untrusted Pointer*, *Cast Pointer*, or *Forbidden Address* (including *Null Pointer*) would break pointer safety and lead to final errors such as *Object Corruption*, *Memory Leak*, *Type Confusion*, and *NULL Pointer Dereference*, respectively. Subsequently, bugs and faults covered by any non-_MEM classes, whose operations produce such pointers, should also be avoided.

According to the proper memory-related operation flow (see Fig. 9), a pointer may be initialized before or after the allocation of its object. However, it must be initialized before it is used to address its object, repositioned after the reallocation of its object, and reassigned after the deallocation of its object. These correspond to the MAD *Wild Pointer* and *Dangling Pointer* errors. If an object is reallocated because of being extended or reduced, all of its owners must be repositioned.

An object must not be read before it is initialized (i.e., the first write) and must be cleared (i.e., the last write) before it is deallocated. An unneeded object must be cleared and deallocated, and all of its pointers (owners) must be reassigned. It should not be possible to access and use its data after it is deallocated. These correspond to the BF MUS *Uninitialized Object* error and *Not Cleared Object*, *Memory Leak*, and *Use After Deallocate* (i.e., use after free or use after return) final errors. Memory leaks enable IEX and DOS failures. The use of deallocated objects enables IEX, TPR, DOS, and ACE failures.

Temporal memory safety may depend on input data safety. For example, verification of an input size toward the actual size of a buffer before *Read* or *Write* will eliminate related buffer overflows. Spatial memory safety may depend on data type safety. For example, avoiding *Cast*-related data errors that may lead to the use of an incorrect element size and *Reposition* overbound or underbound will eliminate related buffer overflows.

Avoiding the meaningful ⟨*bug/fault*, *operation*⟩ couples of the BF _MEM classes would guarantee temporal and spatial memory safety, as well as general pointer safety. Avoiding erroneous pointer arithmetics will also eliminate related buffer overflows. For example, proper type declaration would avoid type coercion at argument passing to functions that calculate the size of a buffer and result in flipped signs and wrap-around (e.g., integer overflows), rounded (i.e., breaking floating point safety), or truncated values.

### 9.3. Data Type Safety

*Data type safety* ensures the proper use of entities (e.g., objects, functions, and data types) in code. That is, objects, functions, and data types are properly declared, defined, and referenced; objects are properly typecast or coerced; and functions are correctly called to perform error-free type-related computations. It is addressed by the BF Data Type Bugs Model (see Fig. 10) operation flow and the BF DCL, NRS, TCM, and TCV classes (see BF _DAT at [1]) that define why, where, and how data type safety could break.

Data type safety has both temporal and spatial aspects. *Temporal data type safety* ensures the use of data values that are compatible and data types that are non-confused with the declared data type of an object. Entities must not only be declared but also defined and their names properly resolved and bound. Compute and evaluate functions must also be defined with the appropriate argument data types. Temporal data type safety is mostly covered by the type system of the programming language. However, DCL bugs that cause errors (e.g., *Missing Overloaded Function*) or propagate to faults (e.g., *Wrong Argument Type*) could still break data type safety.

*Spatial data type safety* ensures proper object type conversions and use of its layout. It is addressed by the operations that affect the interpretation of the object layout or the elements' size along the y-axis of the _DAT Bugs Model (see Fig. 10): TCV *Cast* and *Coerse*.

Examples of spatial data type safety are cast pointer safety, coerced object safety, and subtype safety. The first prevents a pointer and its object from having incompatible data types. A declared overloaded function must have implementations for all of the needed argument types. Otherwise, *Coerce* will be forced on the argument values. The last example prevents a pointer and its object from having confused data types. A cast pointer can cause different element size interpretations and overall object size that may lead to buffer overflows. A wrong argument type coercion can result in truncated or rounded data values. Downcasting a pointer to a sibling class can cause *Subtype Confusion* and enable an ACE or RCE (of functions of the sibling class) failure.

Data type safety may depend on input data safety. For example, verification of the target data type toward the object (source) data type before the *Cast* of an object pointer will eliminate related *Cast Pointer* errors and *Type Confusion* final errors.

Avoiding the meaningful ⟨*bug/fault*, *operation*⟩ couples of the BF _DAT classes would guarantee temporal and spatial data type safety.

The methodology for the definition of secure coding principles by software, firmware, or hardware execution phases involves the following seven steps (also see Fig. 22):

1. BF Bugs Model: Identify the BF Bugs Model that corresponds to the execution phases for which secure coding principles are to be defined.
2. Temporal and Spatial Operations: Determine the operations along the x-axis of the Bugs Model that relate to the temporal safety for the BF class type. If a y-axis exists, determine which operations relate to its spatial safety for the BF class type.
3. Operations Flow Rules: Formally describe the proper operations flow according to the semantic graphs of the BF Bugs Model.
4. BF Class Type: Identify the BF Class Type that corresponds to the BF Bugs Model and the semantic matrices for each of its BF classes.
5. Spatial Safety Rules: Formally describe what (i.e., bugs or faults) how (i.e., errors and final errors) could break the code safety via spatial operations according to the within weakness causation semantic rules of the BF classes.
6. Temporal Safety Rules: Formally describe what (i.e., bugs or faults) and how (i.e., errors and final errors) could break the code safety via temporal operations according to the within weakness causation semantic rules of the BF classes.
7. Dependency Rules: Identify code safety dependencies according to the BF between weaknesses causation and propagation semantic rules.

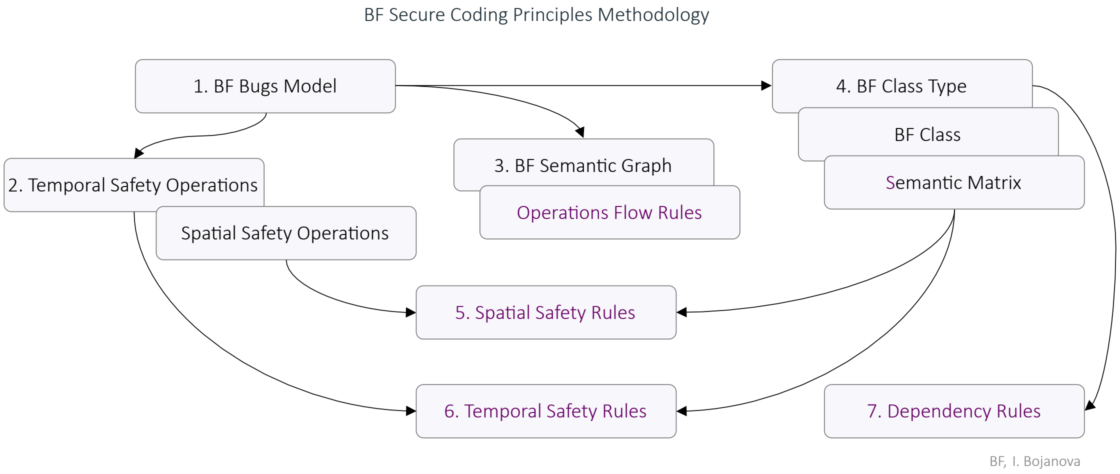


**Fig. 22.** BF secure coding principles methodology

While the BF formal language is descriptive, the secure coding principles are prescriptive against bugs and faults per operation that break specific kinds of code safety.

For more secure coding principles and details, refer to the forthcoming SP 800-231F, *Bugs Framework: Secure Coding Principles*.

# 10. BF Tools

The BF features generation tools that reflect the BF taxonomy, models, and formal language syntax and semantics. The BFCWE tool and the BFCVE tool facilitate the generation of formal weakness and vulnerability specifications. The BF tool guides the creation of complete BF vulnerability specifications. The related BF APIs at [1] provide BF data retrieval and specific tool functionalities.

The BFDB database hosts the BF data. The BF taxonomy structure and semantics rules are organized via a relational database with graph features and via XML and JSON data interchange formats (query them via the BF API at [1]). The BFDB contains the types, names, and definitions of the BF taxons, their relationships within the taxonomy, and the BF weakness and vulnerability semantic relation and propagation matrices and operation flow graphs. The BF mashup database organizes additional data for querying BF toward the CWE, CVE, NVD, GitHub [37], KEV [6], and Exploit Prediction Scoring System (EPSS) [38].

## 10.1. BFCWE Tool

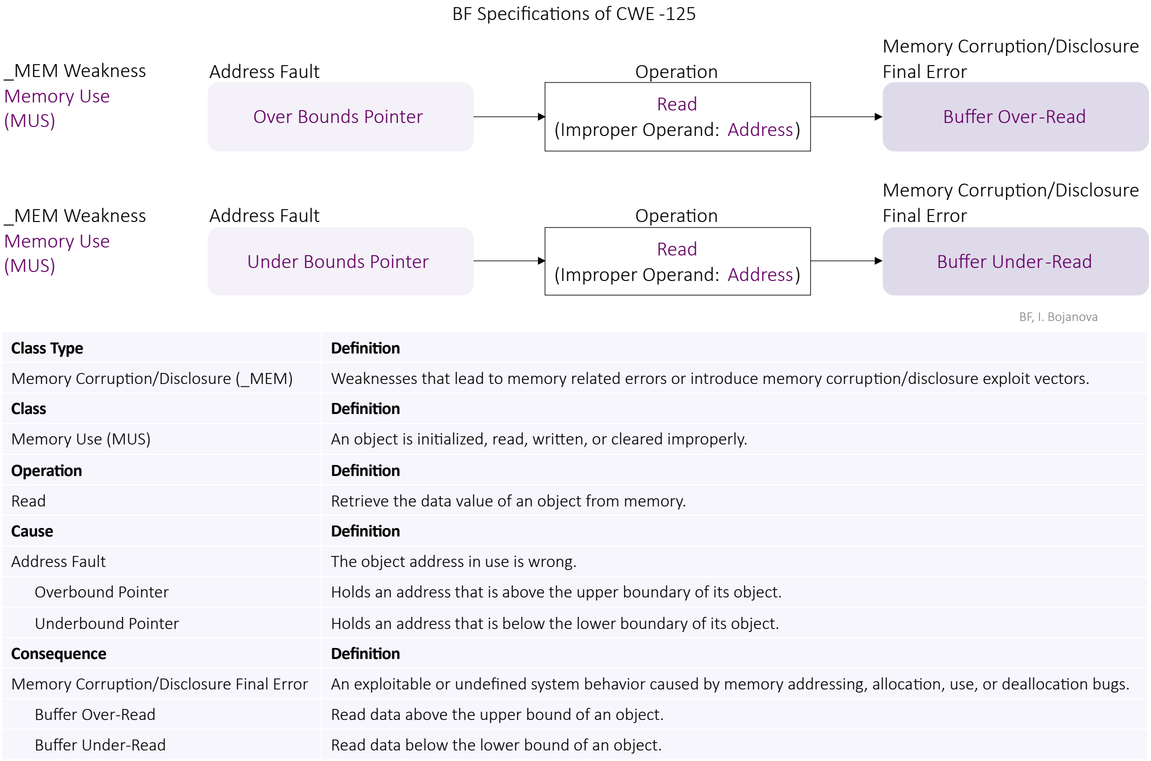
The BFCWE tool facilitates the creation of CWE-to-BF (CWE2BF) mappings by weakness operation, error, final error, and possibly entire ⟨*cause*, *operation*⟩→*consequence*) weakness triples [39]. It also generates BFCWE formal specifications as entries of the BFCWE security weakness types dataset and graphical representations of the CWE2BF mappings and the BFCWE specifications to enhance understanding (e.g., see [16–18, 22]).

Meticulous analysis of the natural language descriptions of CWEs, relevant code examples, and descriptions of related CVEs is conducted to create CWE2BF mappings by weakness operation, error, and final error and then by detailed ⟨*bug*, *operation*⟩→*error*, ⟨*fault*, *operation*⟩→*error*, ⟨*bug*, *operation*⟩→*final error*, and ⟨*fault*, *operation*⟩→*final error* weakness triples [40].

The BFCWE tool is utilized to generate the graphical representations of the CWE2BF mappings for enhanced understanding as directed graphs with parent-child CWE relationships. Examples include _INP CWE2BF [16], _MEM CWE2BF [17], and _DAT CWE2BF [18].

Since a specific CWE should be about a single weakness, any parts of its description that reveal possible causing weaknesses are not considered for the BFCWE specification. However, they are considered for the partial BFCVE specifications (see Sec. 11.3 and BFCVE Partial). All identified weakness triples are checked against the BF matrix of valid ⟨*cause*, *operation*⟩→*consequence* within weakness relations, which defines part of the BF formal language semantics. The same methodology helps reveal overlaps among the CWEs, as many of them have the same BF specification — that is, the same BF weakness triple.

The BFCWE tool is utilized to generate the formal BF specification of each weakness as an entry of the BFCWE security weakness dataset and its graphical representation. However, there could be a set of corresponding BF specifications for some CWEs. For example, the natural language descriptions, demonstrative examples, and potential mitigations for CWE-125 reveal the ⟨*Overbound Pointer, Read*⟩→*Buffer Over-Read* and ⟨*Underbound Pointer, Read*⟩→*Buffer Over-Read* possible weakness triples. Subsequently, these are the possible BF specifications for the main weakness of a CVE mapped to CWE-125. Figure 23 shows their generated graphical representation.



| **Class Type** | **Definition** |
|---|---|
| Memory Corruption/Disclosure (_MEM) | Weaknesses that lead to memory related errors or introduce memory corruption/disclosure exploit vectors. |
| **Class** | **Definition** |
| Memory Use (MUS) | An object is initialized, read, written, or cleared improperly. |
| **Operation** | **Definition** |
| Read | Retrieve the data value of an object from memory. |
| **Cause** | **Definition** |
| Address Fault | The object address in use is wrong. |
| Overbound Pointer | Holds an address that is above the upper boundary of its object. |
| Underbound Pointer | Holds an address that is below the lower boundary of its object. |
| **Consequence** | **Definition** |
| Memory Corruption/Disclosure Final Error | An exploitable or undefined system behavior caused by memory addressing, allocation, use, or deallocation bugs. |
| Buffer Over-Read | Read data above the upper bound of an object. |
| Buffer Under-Read | Read data below the lower bound of an object. |

**Fig. 23.** BF specifications of CWE-125

The BFCWE tool is also useful for the generation and analysis of CWE directed graphs by other criteria. For example, see the directed graphs of hardware CWEs and their analysis in NIST IR 8517 [41].

For more details, refer to the forthcoming SP 800-231G, *Bugs Framework: Tools and APIs*.

## 10.2. BFCVE Tool

The BFCVETool facilitates the creation of CVE-to-BF (CVE2BF) mappings by final error and possibly entire ⟨*bug/fault*, *operation*⟩→*final error* weakness triple. It also generates possible chains of weaknesses for a vulnerability (e.g., a CVE) by an identified failure, final error, or entire final weakness; generates possible BFCVE formal specifications and their graphical representations; and identifies CWEs for NVD assignment [42]. Code analysis and the BF graphical user interface (GUI) functionality can be used to identify and complete the unique unambiguous BF vulnerability specification.

The BF relational database, the NVD Representational State Transfer Application Programming Interface (REST API), and the GitHub REST API are utilized to extract CVEs with assigned CWEs for which *Code with Fix* is available. For example, as of June 20, 2024, there are 5 162 CVEs that map to BF _INP [1, 16] weakness triples by CWE, 4 484 CVEs that map to BF _MEM [1, 17], and 629 CVEs that map to BF _DAT [1, 18] for which GitHub diffs are available via the NVD. Other repositories may also provide fix commits and even the code of vulnerable functions (e.g., DiverseVul [43]).

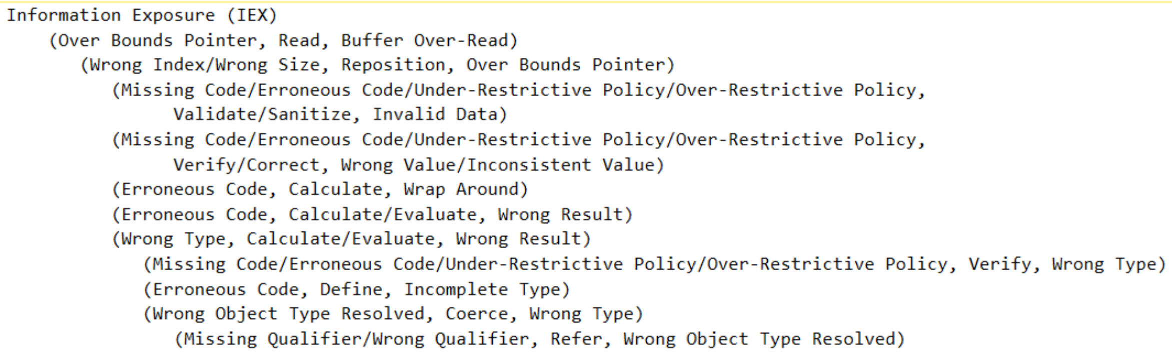
Information on the failure and the final weakness is gained from CVE reports, CVE descriptions, and CWE2BF weakness triple mappings if a CWE is assigned by the NVD. The BFCVE tool utilizes the BFDB relational database and the NVD REST API to extract the CWE2BF triples for that CVE. It then generates CVE2BF mappings by possible final error or final weakness and failure. For a specific CVE, the BFCVE tool applies the BF causation and propagation rules (i.e., the BF formal language syntax and semantics) to go backward from the failure through the final weakness to generate all possible BF chains of weaknesses for that specific CVE independently of whether the CVE *Code with Fix* is available.

Going backward from the failure, the BFCVE tool builds a connected acyclic undirected graph (i.e., a tree whose root is the failure) of all possible weakness chains with type-based backward *fault type*$\curvearrowleft$*error type* match and *fault value*$\curvearrowleft$*error value* propagation or — for weaknesses of the same BF class type — direct match. The chains undergo scrutiny to ensure further alignment with the BF formal language semantics, the causation matrices of meaningful $\langle$*cause*, *operation*$\rangle\rightarrow$*consequence* within weakness relations, the graphs of meaningful (*operation*$_1, \dots,$ *operation*$_n$) bug or fault state paths, and the matrices of meaningful *consequence*$\curvearrowright$*cause* between weaknesses propagations. The identified failure and final weakness triple dramatically reduce the number of generated possible paths in the acyclic graph. This is also a good starting point for specifying vulnerabilities that are not recorded in the CVE.

The CVE *Code with Fix* can then be examined by security researchers or utilizing AI and compared with the generated chains of weakness triples to pinpoint the unique unambiguous BF vulnerability specification. For that, the BF tool functionality and automated code analysis — including via large language models (LLMs) — can be used.

For example, the main vulnerability for CVE-2014-0160 Heartbleed (see Sec. 7.1) is mapped to CWE-125 in the NVD, and the CWE2BF mappings for CWE-125 restrict the final weakness options for Heartbleed to $\langle$*Overbound Pointer*, *Read*$\rangle\rightarrow$*Buffer Over-Read* and $\langle$*Underbound Pointer*, *Read*$\rangle\rightarrow$*Buffer Under-Read* (see Sec. 10.1). However, the CVE-2014-0160 description reveals the word *over*, which indicates that CWE-125 is too abstract for it and eliminates the second final weakness option. In addition, as Heartbleed leads to information exposure, the last part of the BF weaknesses chain is $\langle$*Overbound Pointer*, *Read*$\rangle\rightarrow$*Buffer Over-Read*$\curvearrowright$*IEX*. The *Read* operation uniquely identifies the weakness as an instance of the BF MUS class, as BF classes do not overlap by operation.

Going backward from *Overbound Pointer* via the BF causation and propagation rules, the BFCVE tool generates a tree of suggested weakness chains for Heartbleed (see Fig. 24). The failure is the root, the final weakness is the first node, and a bug weakness is the last node of each path. The only options for the weakness causing the final weakness are ⟨*Wrong Index*, *Reposition*⟩→*Overbound Pointer* and ⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer*. Both have the same options for causing chains, only two of which do not start with a bug, though the preceding weakness options start with a bug. Exhausting these few options via deep code analysis or the use of LLMs would confirm that the unique unambiguous chain for Heartbleed is ⟨*Missing Code*, *Verify*⟩→*Inconsistent Value*↷⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer*↷⟨*Overbound Pointer*, *Read*⟩→*Buffer Over-Read*↷*IEX*.



```
Information Exposure (IEX)
    (Over Bounds Pointer, Read, Buffer Over-Read)
       (Wrong Index/Wrong Size, Reposition, Over Bounds Pointer)
          (Missing Code/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy,
                Validate/Sanitize, Invalid Data)
          (Missing Code/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy,
                Verify/Correct, Wrong Value/Inconsistent Value)
          (Erroneous Code, Calculate, Wrap Around)
          (Erroneous Code, Calculate/Evaluate, Wrong Result)
          (Wrong Type, Calculate/Evaluate, Wrong Result)
             (Missing Code/Erroneous Code/Under-Restrictive Policy/Over-Restrictive Policy, Verify, Wrong Type)
             (Erroneous Code, Define, Incomplete Type)
             (Wrong Object Type Resolved, Coerce, Wrong Type)
                (Missing Qualifier/Wrong Qualifier, Refer, Wrong Object Type Resolved)
```

**Fig. 24.** Generated BF weakness chains for Heartbleed

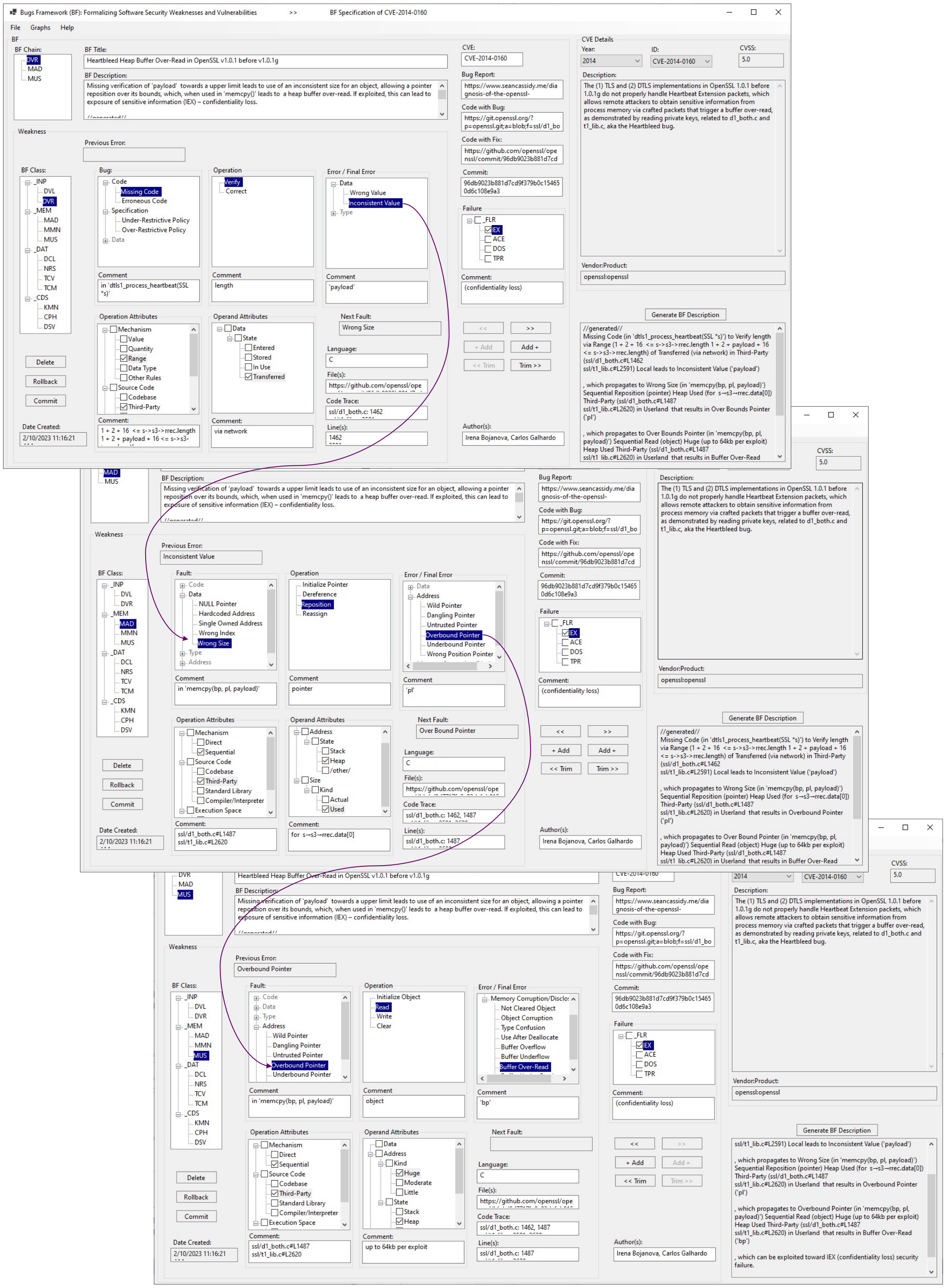
The BFCVE tool generates graphical representations of the BFCVE formal specifications to enhance understanding (see Fig. 21). Related BF tools functionality is the generation of the webpages for the BF class taxonomies (e.g., BF MUS) and the BFCVE specifications (e.g., BF CVE-2014-0160) of the BF website [1].

For more details, refer to the forthcoming SP 800-231G, *Bugs Framework: Tools and APIs*.

### 10.3. BF GUI Tool

The BF tool is a GUI application that works with both the BF relational database and the BF in XML or JSON format; the latter is useful when the database is unavailable [44]. It allows users to create a new BF CVE specification, save it as a machine-readable `.bfcve` file, and open and browse previously created `.bfcve` specifications [22].

The BF tool (see Fig. 25) guides the specification of a security vulnerability as a chain of underlying weaknesses. A security bug causes the first weakness, which leads to an error. This error becomes the cause (i.e., the fault) of the next weakness and propagates through subsequent weaknesses until a final error is reached, enabling a security failure. The causation within a weakness is by a meaningful ⟨*cause*, *operation*⟩→*consequence* relation. The causation between weaknesses is by *error type* to *fault type* match and operation flow or *error*↷*fault* by *value* propagation.



Bugs Framework (BF): Formalizing Software Security Weaknesses and Vulnerabilities >> BF Specification of CVE-2014-0160

BF Chain: DVR, MAD, MUS

BF Title: Heartbleed Heap Buffer Over-Read in OpenSSL v1.0.1 before v1.0.1g

BF Description: Missing verification of 'payload' towards a upper limit leads to use of an inconsistent size for an object, allowing a pointer reposition over its bounds, which, when used in 'memcpy()' leads to a heap buffer over-read. If exploited, this can lead to exposure of sensitive information (IEX) – confidentiality loss.

CVE: CVE-2014-0160

Bug Report: https://www.seancassidy.me/diagnosis-of-the-openssl-

Code with Bug: https://git.openssl.org/?p=openssl.git;a=blob;f=ssl/d1_bo

Code with Fix: https://github.com/openssl/openssl/commit/96db9023b881d7cd

Commit: 96db9023b881d7cd9f379b0c154650d6c108e9a3

CVE Details — Year: 2014 ID: CVE-2014-0160 CVSS: 5.0

Description: 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.

Vendor:Product: openssl:openssl

Failure: □ _FLR ☑ IEX □ ACE □ DOS □ TPR
Comment: (confidentiality loss)

Author(s): Irena Bojanova, Carlos Galhardo

Date Created: 2/10/2023 11:16:21

Weakness

BF Class: _INP (DVL, DVR), _MEM (MAD, MMN, MUS), _DAT (DCL, NRS, TCV, TCM), _CDS (KMN, CPH, DSV)

DVR:
- Previous Error:
- Bug: Code – Missing Code; Comment: in 'dtls1_process_heartbeat(SSL *s)'
- Operation: Verify; Comment: length
- Error / Final Error: Data – Inconsistent Value; Comment: 'payload'
- Operation Attributes: Mechanism □ Value □ Quantity ☑ Range □ Data Type □ Other Rules; Source Code □ Codebase ☑ Third-Party; Comment: 1 + 2 + 16 <= s->s3->rrec.length 1 + 2 + payload + 16 <= s->s3-
- Operand Attributes: Data – State □ Entered □ Stored □ In Use ☑ Transferred; Comment: via network
- Next Fault: Wrong Size
- Language: C
- File(s): https://github.com/openssl/ope
- Code Trace: ssl/d1_both.c: 1462
- Line(s): 1462

MAD:
- Previous Error: Inconsistent Value
- Fault: Data – Wrong Size; Comment: in 'memcpy(bp, pl, payload)'
- Operation: Reposition; Comment: pointer
- Error / Final Error: Address – Overbound Pointer; Comment: 'pl'
- Operation Attributes: Mechanism □ Direct ☑ Sequential; Source Code □ Codebase ☑ Third-Party □ Standard Library □ Compiler/Interpreter; □ Execution Space; Comment: ssl/d1_both.c#L1487 ssl/t1_lib.c#L2620
- Operand Attributes: Address – State □ Stack ☑ Heap □ /other/; Size – Kind □ Actual ☑ Used; Comment: for s→s3→rrec.data[0]
- Next Fault: Over Bound Pointer
- Language: C
- File(s): https://github.com/openssl/ope
- Code Trace: ssl/d1_both.c: 1462, 1487
- Line(s): ssl/d1_both.c: 1487

MUS:
- Previous Error: Overbound Pointer
- Fault: Address – Overbound Pointer; Comment: in 'memcpy(bp, pl, payload)'
- Operation: Read; Comment: object
- Error / Final Error: Memory Corruption/Disclos – Buffer Over-Read; Comment: 'bp'
- Operation Attributes: Mechanism □ Direct ☑ Sequential; Source Code □ Codebase ☑ Third-Party □ Standard Library □ Compiler/Interpreter; □ Execution Space; Comment: ssl/d1_both.c#L1487 ssl/t1_lib.c#L2620
- Operand Attributes: □ Data; Address – Kind ☑ Huge □ Moderate □ Little; State □ Stack ☑ Heap; Comment: up to 64kb per exploit
- Next Fault:
- Language: C
- File(s): https://github.com/openssl/ope
- Code Trace: ssl/d1_both.c: 1462, 1487
- Line(s): ssl/d1_both.c: 1487

Generated BF Description:

//generated//
Missing Code (in 'dtls1_process_heartbeat(SSL *s)') to Verify length via Range (1 + 2 + 16 <= s->s3->rrec.length 1 + 2 + payload + 16 <= s->s3->rrec.length) of Transferred (via network) in Third-Party (ssl/d1_both.c#L1462
ssl/t1_lib.c#L2591) Local leads to Inconsistent Value ('payload')

, which propagates to Wrong Size (in 'memcpy(bp, pl, payload)') Sequential Reposition (pointer) Heap Used (for s→s3→rrec.data[0]) Third-Party (ssl/d1_both.c#L1487
ssl/t1_lib.c#L2620) in Userland that results in Over Bounds Pointer ('pl')

, which propagates to Over Bounds Pointer (in 'memcpy(bp, pl, payload)') Sequential Read (object) Huge (up to 64kb per exploit) Heap Used Third-Party (ssl/d1_both.c#L1487
ssl/t1_lib.c#L2620) in Userland that results in Buffer Over-Read

ssl/t1_lib.c#L2591) Local leads to Inconsistent Value ('payload')

, which propagates to Wrong Size (in 'memcpy(bp, pl, payload)') Sequential Reposition (pointer) Heap Used (for s→s3→rrec.data[0]) Third-Party (ssl/d1_both.c#L1487
ssl/t1_lib.c#L2620) in Userland that results in Overbound Pointer ('pl')

, which propagates to Overbound Pointer (in 'memcpy(bp, pl, payload)') Sequential Read (object) Huge (up to 64kb per exploit) Heap Used Third-Party (ssl/d1_both.c#L1487
ssl/t1_lib.c#L2620) in Userland that results in Buffer Over-Read ('bp')

, which can be exploited toward IEX (confidentiality loss) security failure.

**Fig. 25.** BF GUI tool

If a CVE is being specified, the user can select `CVE Year` and `CVE ID` in the *CVE Details* GroupBox to display its description, vendor, and product from the CVE repository and its CVSS [10, 11] severity score from the NVD. To create a BFCVE specification of that CVE, the user is guided to define an initial weakness, possible propagation weaknesses, and a final weakness leading to a failure. If a vulnerability has only one underlying weakness, it would be both the initial and final weakness.

To start defining a weakness, the user has to select a BF weakness class from the `BF Class` TreeView in the `Weakness` GroupBox container, where the classes are grouped by BF class types as parent nodes. The selection of a class populates the five TreeView controls in the `Weakness` GroupBox container: `Bug/Fault`, `Operand`, `Error/Final Error`, `Operation Attributes`, and `Operand Attributes`. To specify the weakness, the user has to select child nodes from the five TreeView controls and enter comments in the text boxes beneath them.

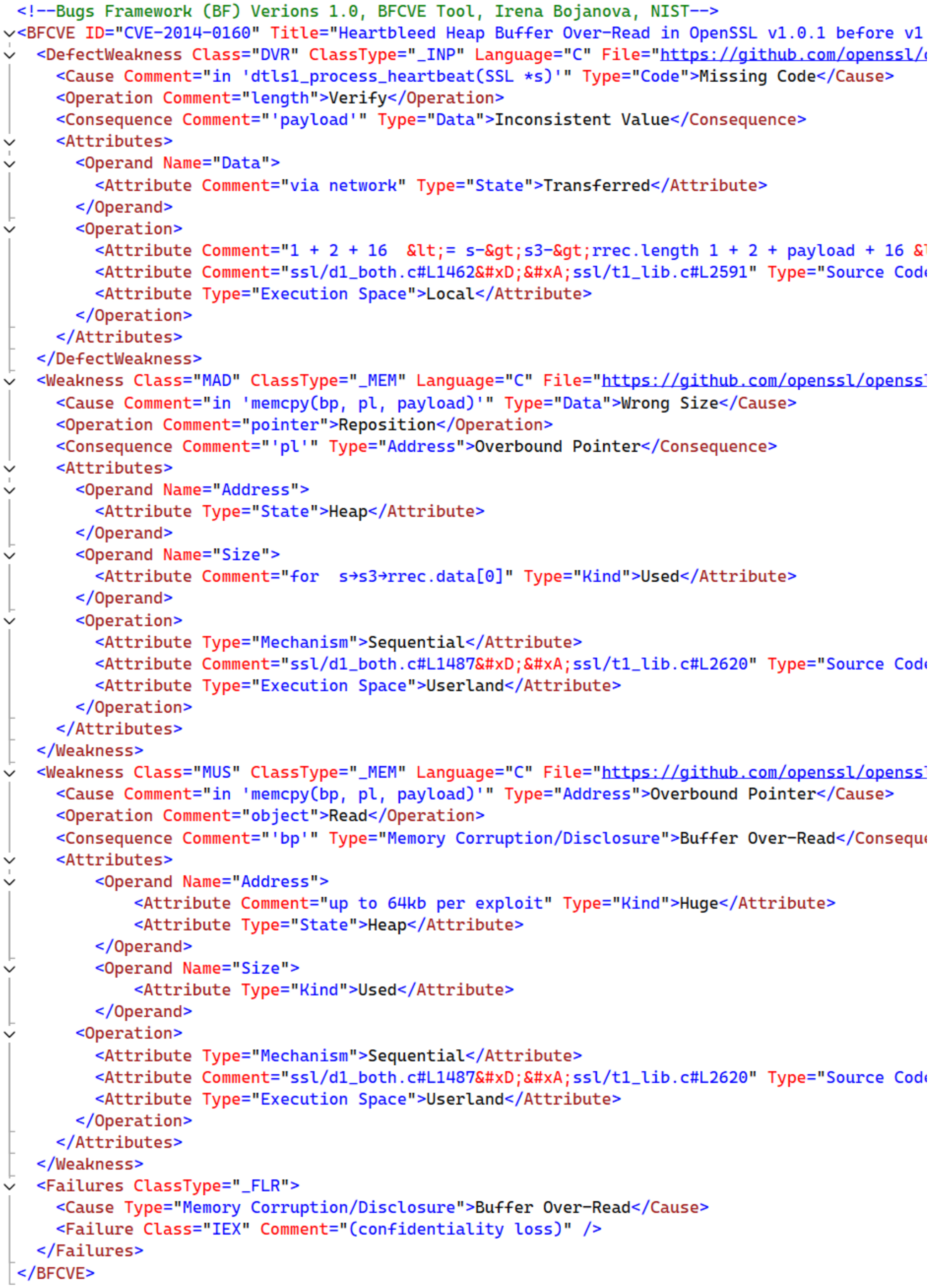
The BF tool can enforce that the initial weakness starts with a bug and the rest of the weaknesses start with a fault. However, this is not necessary for partial specifications or if a vulnerability starts with a hardware defect-induced fault. The `Bug/Fault` label changes to `Bug` when the initial weakness is viewed and to `Fault` when the propagation or final weakness is viewed. In the case of a bug, the child nodes are only allowed under the `Code` and `Specification` nodes. In the case of a fault, the child nodes are only allowed under the `Data`, `Type`, and `Address` nodes. Tooltips with term definitions are displayed over all TreeView nodes. The BF tool also enforces that the weakness with the *final error* consequence is the final weakness leading to a failure.

Once a weakness is specified, the user can proceed via the >> button and create the next weakness of the vulnerability chain. Weakness chaining is restricted by the error-to-fault by type match rule, which — to a large extent — also restricts to meaningful operation flow, as the BF classes are developed to adhere to the BF bugs models that are specific to their BF class types. The `Generate BF Description` button displays a draft BF description based on the selected values from the five TreeView controls and `Comment` text boxes.

Figure 26 presents the BF specification of the main vulnerability chain of Heartbleed in XML format generated by the BF tool. In addition to the XML attributes that relate to the BF Taxonomy, the `CVE-2014-0160.bfcve` also contains generated natural language descriptions, programming language, links to reports, code with bugs, code with fixes, commit IDs, authors, and code locations (i.e., lines) per weakness. For more details, see BF CVE-2014-0160 at [1].

The BF tool demonstrates how the BF taxonomy and causation and propagation rules tie together into the strict BF formal language. It uses the BFCVE tool functionality to generate graphical representations of the BF formal specifications to enhance understanding. For example, refer to the BF CVE-2014-0160 and related BF taxons definitions at [1].

For more details, refer to the forthcoming SP 800-231G, *Bugs Framework: Tools and APIs*.



```
<!--Bugs Framework (BF) Verions 1.0, BFCVE Tool, Irena Bojanova, NIST-->
<BFCVE ID="CVE-2014-0160" Title="Heartbleed Heap Buffer Over-Read in OpenSSL v1.0.1 before v1
  <DefectWeakness Class="DVR" ClassType="_INP" Language="C" File="https://github.com/openssl/
    <Cause Comment="in 'dtls1_process_heartbeat(SSL *s)'" Type="Code">Missing Code</Cause>
    <Operation Comment="length">Verify</Operation>
    <Consequence Comment="'payload'" Type="Data">Inconsistent Value</Consequence>
    <Attributes>
      <Operand Name="Data">
        <Attribute Comment="via network" Type="State">Transferred</Attribute>
      </Operand>
      <Operation>
        <Attribute Comment="1 + 2 + 16  &lt;= s-&gt;s3-&gt;rrec.length 1 + 2 + payload + 16 &
        <Attribute Comment="ssl/d1_both.c#L1462&#xD;&#xA;ssl/t1_lib.c#L2591" Type="Source Cod
        <Attribute Type="Execution Space">Local</Attribute>
      </Operation>
    </Attributes>
  </DefectWeakness>
  <Weakness Class="MAD" ClassType="_MEM" Language="C" File="https://github.com/openssl/openss
    <Cause Comment="in 'memcpy(bp, pl, payload)'" Type="Data">Wrong Size</Cause>
    <Operation Comment="pointer">Reposition</Operation>
    <Consequence Comment="'pl'" Type="Address">Overbound Pointer</Consequence>
    <Attributes>
      <Operand Name="Address">
        <Attribute Type="State">Heap</Attribute>
      </Operand>
      <Operand Name="Size">
        <Attribute Comment="for  s→s3→rrec.data[0]" Type="Kind">Used</Attribute>
      </Operand>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Comment="ssl/d1_both.c#L1487&#xD;&#xA;ssl/t1_lib.c#L2620" Type="Source Cod
        <Attribute Type="Execution Space">Userland</Attribute>
      </Operation>
    </Attributes>
  </Weakness>
  <Weakness Class="MUS" ClassType="_MEM" Language="C" File="https://github.com/openssl/openss
    <Cause Comment="in 'memcpy(bp, pl, payload)'" Type="Address">Overbound Pointer</Cause>
    <Operation Comment="object">Read</Operation>
    <Consequence Comment="'bp'" Type="Memory Corruption/Disclosure">Buffer Over-Read</Consequ
    <Attributes>
        <Operand Name="Address">
            <Attribute Comment="up to 64kb per exploit" Type="Kind">Huge</Attribute>
            <Attribute Type="State">Heap</Attribute>
        </Operand>
        <Operand Name="Size">
            <Attribute Type="Kind">Used</Attribute>
        </Operand>
      <Operation>
        <Attribute Type="Mechanism">Sequential</Attribute>
        <Attribute Comment="ssl/d1_both.c#L1487&#xD;&#xA;ssl/t1_lib.c#L2620" Type="Source Cod
        <Attribute Type="Execution Space">Userland</Attribute>
      </Operation>
    </Attributes>
  </Weakness>
  <Failures ClassType="_FLR">
    <Cause Type="Memory Corruption/Disclosure">Buffer Over-Read</Cause>
    <Failure Class="IEX" Comment="(confidentiality loss)" />
  </Failures>
</BFCVE>
```

**Fig. 26.** BF Heartbleed in XML

## 11. BF Datasets and Systems

The BF formalism enables systematic comprehensive labeling of common weakness types (including CWEs) and publicly disclosed vulnerabilities (including CVEs). The BF tools and APIs enable the generation of weakness and vulnerability specifications.

As of June 20, 2024, 31 % of the CWEs and 64 % of the CVEs labeled by the NVD with CWEs map to BF _INP, _MEM, and _DAT classes. These provide a solid base for the creation of comprehensively labeled BFCWE weakness and BFCVE vulnerability datasets.

The BF and the continuous development of BFCWE and BFCVE datasets would allow for multidimensional representations of vulnerabilities in contrast to the one-dimensional representation provided by the CVE enumeration [8].

### 11.1. BFCWE Dataset

There are 938 CWEs [3] as of June 20, 2024. Of those, 157 map to the BF _INP [1, 16] class type; 60 map to BF _MEM [1, 17]; and 72 map to BF _DAT [1, 18]. These 289 unique CWEs form 31 % (see Fig. 27) of the CWE repository and provide the basis for the systematic creation of a comprehensively labeled BFCWE weakness dataset. Most of them represent the most dangerous weakness types by BF final error: *Injection* and *Memory Corruption/Disclosure* [45].

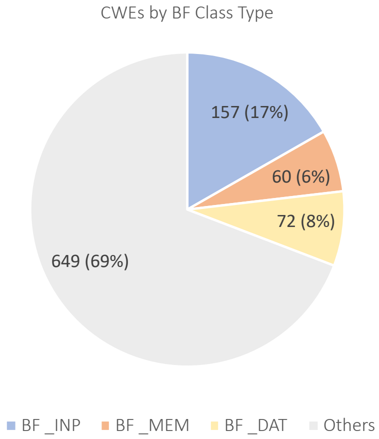


**Fig. 27.** CWEs by BF class types

The NVD uses the 130 “most commonly seen weaknesses” from CWE View-1003 [46] to label CVEs but may also list other CWEs assigned by third-party contributors. The BFCWE dataset may cover software, firmware, or hardware weakness types that are not listed in the CWE.

The methodology for the creation of a BFCWE dataset that utilizes the BF formal language involves the following four steps:

1. CWEs: Identify CWEs and other weakness types that correspond to a specific software, firmware, or hardware execution phase.
2. CWE2BF Mappings: Create CWE2BF mappings by BF operation, error, final error, and detailed ⟨*bug/fault*, *operation*⟩→*error/final error* weakness triples [16–18, 22].
3. BF Specifications: Generate BFCWE formal specifications as entries of the BFCWE security weakness types dataset.
4. Graphical Representations: Generate BFCWE graphical representations to enhance understanding of the CWE2BF mappings by operation, error, final error, and complete weakness triples with parent-child CWE relations.

As the BFCWE specifications are essentially partial BFCVE specifications, the matrix and dataset are also continuously enriched by newly developed BF specifications of CVEs and other reported security vulnerabilities. All developed BFCWE specifications are added to the comprehensively labeled BFCWE dataset (query it via the BFCWE API at [1]).

The BFCWE dataset augments the NVD (see [47]) and the CWE via formal BF specifications of common weaknesses as BF weakness triples and severity-related attributes. However, the BF has the expressive power to clearly describe any security weakness, not only the types listed in the CWE.

### 11.2. BFCVE Dataset

There are over 180 472 CVEs labeled with CWEs by the NVD [5] as of June 20, 2024. Of those, 68 513 map to the BF _INP [1, 16] class type by final error, 46 231 map to _MEM [1, 17], and 3 631 map to _DAT [1, 18] (see Fig. 28). These 118 375 unique CVEs represent 64 % of the CVEs labeled with CWEs and provide the basis for the systematic creation of a comprehensively labeled BFCVE vulnerability dataset. Most of them relate to the most dangerous weakness types by BF final error: *Injection* and *Memory Corruption/Disclosure* [45].

The methodology for the creation of a BFCVE dataset that utilizes the BF formal language involves the following nine steps:

1. CVEs with Code: Query the NVD and other vulnerability repositories for CVEs and other vulnerabilities with available GitHub commits — that is, CVEs for which *Code with Fix* is available.
2. Failure Mapping: Analyze each CVE to determine the reported failures, and map them to BF *Failure* classes.

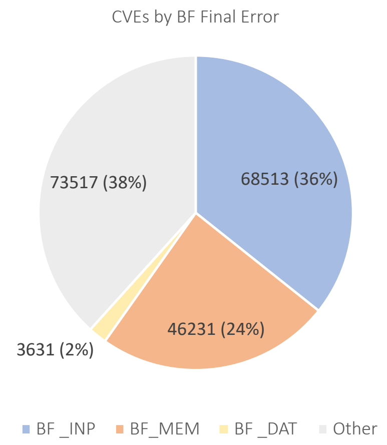


**Fig. 28.** CVEs by BF class types

3. Final Error Mapping: Analyze each CVE to determine the reported sink — in some cases, it is also the root cause or wrongly reported as such — and map it to a BF *Final Error* consequence.

4. CVE2BF Mappings: Utilize steps from the methodology for the creation of BFCWE, and create CVE2BF mappings by final weakness and failure for CVEs with assigned CWEs for which *Code with Fix* is available.

5. Backward State Tree: Generate possible backward chains of weaknesses for a vulnerability by its identified failure and some or all of the elements of the final ⟨*fault*, *operation*⟩→*final error* weakness or — in the case of a one-weakness vulnerability — ⟨*bug*, *operation*⟩→*final error* weakness (see Sec. 3.4).

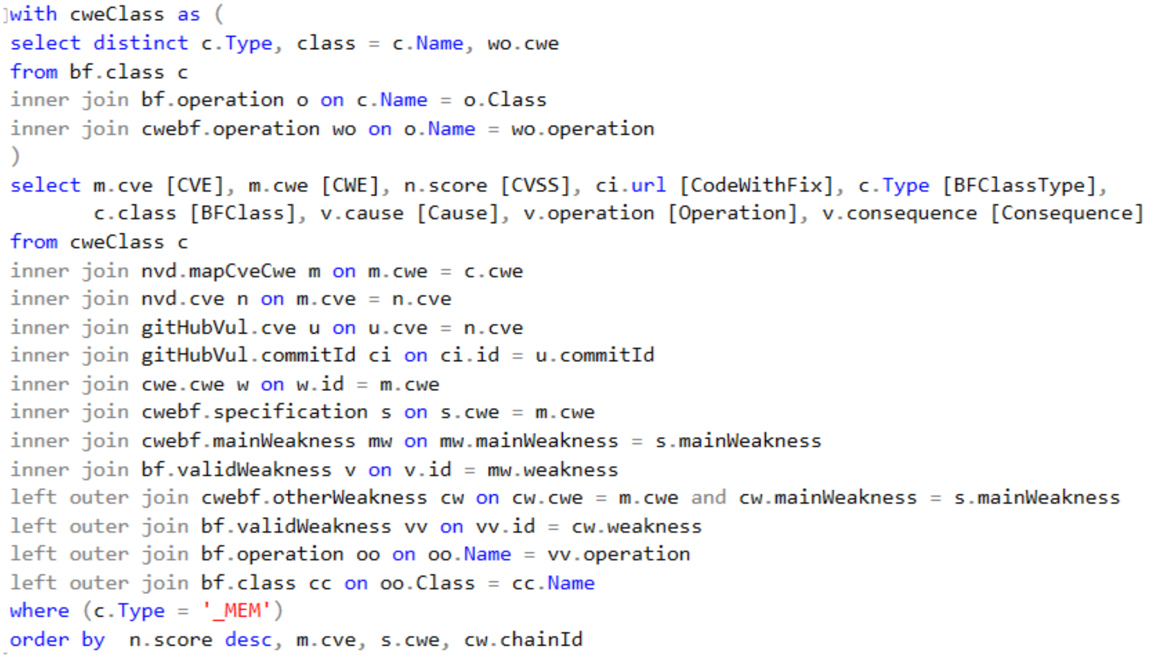
6. Bug or Fault Location: Identify where in the code (i.e., file and lines) the resolved bug or a mitigated fault happened. Comparison of available *Code with Bug* and *Code with Fix* commits would help identify these locations. Improper operation flow by BF bugs models would reveal missing operations (i.e., *Missing Code bugs*).

7. BF Specifications: Conduct deep code analysis — including via LLMs — to filter the generated chains, and use the BF formal language to complete the unambiguous BF vulnerability specifications.

8. Graphical Representations: Generate BFCVE graphical representations to enhance understanding of the BF vulnerability specifications as entries for the BFCVE security vulnerability dataset.

9. CWE Assignments: Identify, refine, and recommend CWEs for NVD assignment. Although this step may seem illogical since a BF specification already provides comprehensive information, it may be useful when comparing CWE-based testing tool reports or if a more appropriate CWE is identified.

A key part of the BFCVE dataset generation is the use of preliminary sets of partial BF specifications of CVEs for which *Code with Fix* is available. These CVE sets are generated by querying the NVD and specific GitHub repositories toward the BFDB. For example, Fig. 29 shows a SQL query for vulnerabilities related to the BF _MEM class type [1, 17] toward a repository with fix commits that are extracted via the GitHub REST API. The query also identifies the possible BF chains of weaknesses for each vulnerability. Security experts and LLMs can then conduct deep code analysis to create the complete BF vulnerability specifications.



```
with cweClass as (
select distinct c.Type, class = c.Name, wo.cwe
from bf.class c
inner join bf.operation o on c.Name = o.Class
inner join cwebf.operation wo on o.Name = wo.operation
)
select m.cve [CVE], m.cwe [CWE], n.score [CVSS], ci.url [CodeWithFix], c.Type [BFClassType],
       c.class [BFClass], v.cause [Cause], v.operation [Operation], v.consequence [Consequence]
from cweClass c
inner join nvd.mapCveCwe m on m.cwe = c.cwe
inner join nvd.cve n on m.cve = n.cve
inner join gitHubVul.cve u on u.cve = n.cve
inner join gitHubVul.commitId ci on ci.id = u.commitId
inner join cwe.cwe w on w.id = m.cwe
inner join cwebf.specification s on s.cwe = m.cwe
inner join cwebf.mainWeakness mw on mw.mainWeakness = s.mainWeakness
inner join bf.validWeakness v on v.id = mw.weakness
left outer join cwebf.otherWeakness cw on cw.cwe = m.cwe and cw.mainWeakness = s.mainWeakness
left outer join bf.validWeakness vv on vv.id = cw.weakness
left outer join bf.operation oo on oo.Name = vv.operation
left outer join bf.class cc on oo.Class = cc.Name
where (c.Type = '_MEM')
order by  n.score desc, m.cve, s.cwe, cw.chainId
```

**Fig. 29.** NVD-GitHub-BF query for _MEM CVEs

A similar NVD-GitHub-BF query is used to generate the BFCVE Partial dataset of CVEs for which GitHub *Code with Fix* is available. As of June 20, 2024, there are 5 162 BF _INP CVEs with GitHub commits in NVD, 4 484 _MEM CVEs, and 629 _DAT CVEs.

This methodology would also guide the creation of BF specifications of vulnerabilities for which code is not available, and insights from existing BF specifications would contribute to their analyses. Going backward from a final weakness would reveal options for previous weaknesses until a weakness with a bug as a cause is reached. For example, going backward from ⟨*Wrong Size*, *Reposition*⟩→*Overbound Pointer* reveals that the previous causing weakness is a BF Data Validation (DVL) initial weakness among ⟨*Missing Code / Erroneous Code / Under-Restrictive Policy / Over-Restrictive Policy*, *Verify / Correct*⟩→*Wrong Value / Inconsistent Value*.

Developed BFCVE specifications are added to the comprehensively labeled BFCVE dataset (query it via the BFCVE API at [1]). The BF semantic matrices, graphs, and datasets are also continuously enriched by the newly developed formal BF specifications of CVEs and other reported security vulnerabilities.

The BFCVE dataset augments the NVD (see [47]) and CVE via formal BF specifications of the publicly disclosed vulnerabilities as chains of weaknesses. However, the BF has the expressive power to clearly describe any security weakness and vulnerability, not only those listed in the CWE and CVE. It has its own databases with causal weakness taxonomies and formal vulnerability specifications composed of underlying weaknesses specifications.

For more details, refer to the forthcoming SP 800-231H, *Bugs Framework: Datasets and Classifications*.

### 11.3. BF Vulnerability Classifications

The BF Vulnerability Classification Model (see Fig. 30) defines how the BF taxonomy and tools are utilized to generate BFCWE and BFCVE datasets (see Sec. 11) and query them and possibly other vulnerability-related repositories to create the BFVul dataset of diverse multidimensional vulnerability classifications based on common properties and similarities.

The methodology for the creation of BF-based vulnerability classifications involves the following steps:

1. BFCWE Dataset: Create a comprehensively labeled weakness dataset.
2. BFCVE Dataset: Create a comprehensively labeled vulnerability dataset.
3. Severity: Query the CVE for CVSS scores, or use other automated approaches to determine the vulnerability severity score.
4. Platform: Query the CVE for associated CPEs.
5. Exploitation: Query the NVD and EPSS for the probability of a CVE being exploited in the next 30 days.
6. Priority: Query the NVD and KEV or use other automated approaches to determine prioritization for remediation.
7. Vulnerability Classifications: Generate multidimensional vulnerability classifications based on common properties and similarities.

Security vulnerabilities could be classified by common root causes (i.e., software or firmware bugs or hardware defect-induced bugs or faults), such as declaring a variable of a wrong data type. They could also be classified by any other BF taxons, such as propagating faults, common final errors, operation and operand attributes, identical BF specifications (i.e., chains of weaknesses), and even the number of underlying weaknesses.

The BF operation and operand attributes provide insight into the severity of the weaknesses and how they relate to commonly used scores, such as CVSS [11] and EPSS [38].

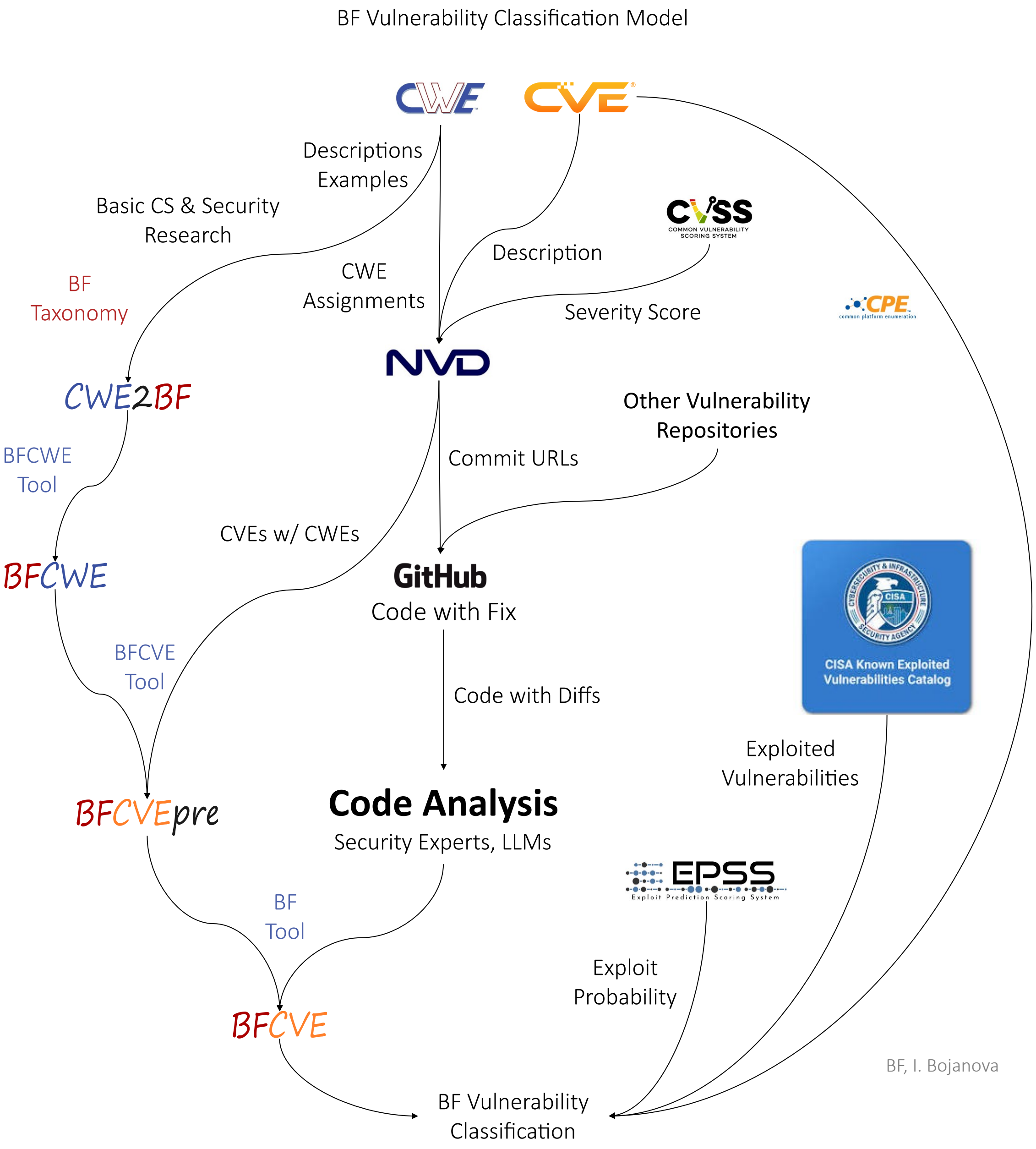


**Fig. 30.** BF Vulnerability Classification Model

Their analysis would allow for deeper research on the most significant [45] and most exploited [48] weaknesses and vulnerabilities. Intriguing classifications by BF classes and CPE [49] data may reveal systematic input/output check safety, memory safety, data type safety, and other secure coding problems by particular vendors and products.

These multidimensional BF vulnerability classifications (query some of them via the BFVul API at [1]) would contribute to a deeper analysis and refined understanding of security weaknesses, vulnerabilities, exploits, and failures. They would enable more focused cybersecurity research and the highly informed development of effective countermeasures against potential security threats and specific exploits.

For more details, refer to the forthcoming SP 800-231H, *Bugs Framework: Datasets and Classifications*.

### 11.4. BF Systems

The BF supports the development of diverse systems related to bug identification and triaging, vulnerability detection, analysis, prioritization, reporting, and resolution or mitigation. The methodology for the development of a BF-based system involves the following steps:

1. Bug Identification: Utilize steps from the methodology for the creation of BFCVE and BFVul datasets, and identify and label the *root cause* of the vulnerability.
2. Vulnerability Detection: Utilize steps from the methodology for the creation of BFCVE and BFVul datasets, and identify and label the *weaknesses* underlying the vulnerability. This may include automated analysis via static and dynamic code analysis tools, and simulation or emulation algorithms that reflect the BF methodologies. Given the formal specification of code and the BF definitions of weakness, vulnerability, and failure, formal methods may also be applied to detect vulnerabilities.
3. Report Generation: Utilize steps from the methodology for the creation of BFCVE and BFVul datasets, and generate a BF formal specification, natural language description, and machine-readable and graphical representations of the vulnerability.

   An LLM may also be prompted to generate the report for that CVE given a CVE description, examples, reports, other references, the code with bug, the code with fix, BF security concept definitions, machine-readable representations of BF taxonomies (including definitions for the taxons and taxon types), and exemplary BF specifications (i.e., entries from the BFCVE dataset).
4. Severity and Prioritization: Determine the vulnerability severity score, and assess whether it needs to be resolved or mitigated urgently. This would be based on the BF-labeled weaknesses and operation and operand *attributes* per weakness and may include analysis of data from services and repositories, such as the EPSS [38], CVSS [11], and KEV [6].
5. Resolution: Determine how the vulnerability should be resolved based on fixing the identified and BF-labeled *bug* of the vulnerability chain or more than one bug in the cases of converging vulnerabilities.
6. Mitigations: Determine the possible ways to mitigate the detected vulnerability based on fixing one of the BF-labeled *faults* through the vulnerability chain.

   For more details, refer to the forthcoming SP 800-231I, *Bugs Framework: Systems*.

## 12. Conclusion

This Special Publication presents an overview of the Bugs Framework (BF) [1] systematic approach and methodologies for the classification of bugs and faults per orthogonal by operation execution phases, formal specification of weaknesses and vulnerabilities, definition of secure coding principles, generation of comprehensively labeled weakness and vulnerability datasets and vulnerability classifications, and development of BF-based algorithms and systems.

The BF weakness and failure taxonomies and bugs and vulnerability models form the basis for the BF ACFG that generates the BF formal language. The BF also helps formally define secure coding principles, such as input/output check safety, memory safety, and data type safety. The BF formal language is descriptive in that it is used to formally specify encountered or predicted weaknesses and vulnerabilities. The BF secure coding principles are prescriptive in that they prevent the bugs and faults per operation that break specific related kinds of code safety.

The BF formalism supports a deeper understanding of vulnerabilities as chains of weaknesses and allows for backward bug identification from a failure. It enables the development of new static and dynamic analysis, simulation, and emulation algorithms (e.g., see [2]). AI or formal methods-enabled capabilities could be used to identify bugs and detect, analyze, prioritize, and resolve or mitigate vulnerabilities (i.e., fix the bug or a fault of each vulnerability, respectively) to secure critical infrastructure and supply chains.

The weakness and vulnerability BF specification datasets augment the CWE, CVE, and NVD. However, the BF has the expressive power to clearly describe any other security weaknesses and vulnerabilities. It also allows for the prediction and identification of as yet unencountered security weakness types, which allows for the prediction and detection of new kinds of vulnerabilities.

The BF aims to become the new standard for the specification and labeling of security weaknesses and vulnerabilities. It enables the clear and precise expression of security bugs, weaknesses, vulnerabilities, and failures. Government institutions could improve the descriptions in public vulnerability repositories and create advanced policies and guidelines for software, firmware, and hardware testing. Security companies could improve their testing tools and bug and vulnerability reports. Academics could teach better about security bugs, weaknesses, and vulnerabilities and conduct deeper security vulnerability and failure research. All of these would lead to unambiguous communication about cybersecurity, the increased precision of code review tools, and a decrease in security bugs, weaknesses, and vulnerabilities.

## References

[1] Irena Bojanova (2014-2024) NIST Bugs Framework (BF) Website. Available at https://usnistgov.github.io/BF.

[2] Kedrian James, K Valakuzhy, K Snow, F Monrose (June 2024) CrashTalk: Automated Generation of Precise, Human Readable, Descriptions of Software Security Bugs. *CODASPY '24: Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*, pp 337 – 347. https://doi.org/10.1145/3626232.3653256

[3] MITRE (2006-2024) Common Weakness Enumeration (CWE). Available at https://cwe.mitre.org.

[4] MITRE (1999-2024) Common Vulnerabilities and Exposures (CVE). Available at https://cve.mitre.org.

[5] NIST (1999-2024) National Vulnerability Database (NVD). Available at https://nvd.nist.gov.

[6] CISA (2021-2024) Known Exploited Vulnerabilities Catalog (KEV). Available at https://www.cisa.gov/known-exploited-vulnerabilities-catalog.

[7] MITRE CWE History. Available at https://cwe.mitre.org/about/history.html.

[8] David E Mann, S M Christey (Jan. 8, 1999) Towards a Common Enumeration of Vulnerabilities. Available at https://www.cve.org/Resources/General/Towards-a-Common-Enumeration-of-Vulnerabilities.pdf.

[9] MITRE CVE History. Available at https://www.cve.org/About/History.

[10] Peter Mell, K Kent, S Romanosky (2006) Common vulnerability scoring system. Available at https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=50899.

[11] FIRST (2015-2024) Common vulnerability scoring system special interest group. Available at https://www.first.org/cvss.

[12] Irena Bojanova (Dec. 9, 2014) Formalizing Software Bugs. *NIST, ITL, SSD*. Available at https://www.nist.gov/publications/formalizing-software-bugs.

[13] Irena Bojanova (Apr. 8, 2015) Towards a 'Periodic Table' of Bugs. *NIST, ITL, SSD*. Available at https://www.nist.gov/publications/towards-periodic-table-bugs.

[14] Yan Wu, I Bojanova, Y Yesha (2015) They Know Your Weaknesses - Do You?: Reintroducing Common Weakness Enumeration. *CrossTalk (The Journal of Defense Software Engineering)*, pp 44–50. Available at https://web.archive.org/web/20180425211828id_/http://static1.1.sqspcdn.com/static/f/702523/26523304/1441780301827/201509-Wu.pdf?token=WJEmDLgmpr3rIZHriubA20L%2F1%2F4%3D.

[15] Irena Bojanova, P E Black, Y Yesha, Y Wu (October 2016) The NIST Bugs Framework (BF): A Structured Approach to Express Bugs. *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp 175–182. https://doi.org/10.1109/QRS.2016.29

[16] Irena Bojanova, C E Galhardo (October 2021) Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp 111–120. https://doi.org/10.1109/ISSREW53611.2021.00052

[17] Irena Bojanova, C E Galhardo (July 2021) Classifying Memory Bugs Using Bugs Framework Approach. *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp 1157–1164. https://doi.org/10.1109/COMPSAC51774.2021.00159

[18] Irena Bojanova, C E Galhardo, S Moshtari (October 2022) Data Type Bugs Taxonomy: Integer Overflow, Juggling, and Pointer Arithmetics in Spotlight. *2022 IEEE 29th Annual Software Technology Conference (STC))*, pp 192–205. https://doi.org/10.1109/STC55697.2022.00035

[19] Constable S (Febryary 29, 2024) Chips & Salsa: Industry Collaboration for new Hardware CWEs. Available at https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Chips-Salsa-Industry-Collaboration-for-new-Hardware-CWEs/post/1575521.

[20] Kedrian James, Y Du, S Das, F Monrose (December 2022) Separating the Wheat from the Chaff: Using Indexing and Sub-Sequence Mining Techniques to Identify Related Crashes During Bug Triage. *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pp 31–42. https://doi.org/10.1109/QRS57517.2022.00014

[21] Drew Malzahn, Z Birnbaum, C Wright-Hamor (2020) Automated Vulnerability Testing via Executable Attack Graphs. *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)* (IEEE), p 1–10. https://doi.org/10.1109/CyberSecurity49315.2020.9138852

[22] Irena Bojanova (Jan.-Feb. 2024) Comprehensively Labeled Weakness and Vulnerability Datasets via Unambiguous Formal NIST Bugs Framework (BF) Specifications. *IEEE IT Professional*, Vol. 26, 1, pp 60–68. https://doi.org/10.1109/MITP.2024.3358970

[23] Wikipedia (2023) Heartbleed. Available at https://en.wikipedia.org/wiki/Heartbleed.

[24] IEEE Computer Society (September, 1990) IEEE Standard 610.12-1990, Glossary of Software Engineering Terminology. Available at https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=159342.

[25] CISA (2022) ICS Advisory (ICSA-21-119-04). Available at https://www.cisa.gov/uscert/ics/advisories/icsa-21-119-04.

[26] Omri Ben-Bassat TA (2021) ERROR: BadAlloc! - Broken Memory Allocators Led to Millions of Vulnerable IoT and Embedded Devices. Blackhat USA 2021 Available at https://www.youtube.com/watch?v=lSvygMc8uc0.

[27] Assane Gueye, C E Galhardo, I Bojanova (Jul.-Aug. 2023) Critical Software Security Weaknesses. *IEEE IT Professional*, Vol. 25, 4, pp 11–16. https://doi.org/10.1109/MITP.2023.3297387

[28] OpenSSL (2005) Openssl/ssl/d1_both.c. Available at https://git.openssl.org/?p=openssl.git;a=blob;f=ssl/d1_both.c;h=0a84f957118afa9804451add380eca4719a9765e;hb=4817504d069b4c5082161b02a22116ad75f822b1.

[29] Sean Cassidy (April, 2014) Diagnosis of the OpenSSL Heartbleed Bug. Available at https://www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html.

[30] OpenSSL (2014) openssl/openssl, openssl_1_0_1-stable. Available at https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3.

[31] ISO/IEC 14977 (1996) International Standard: Information technology — Syntactic metalanguage — Extended BNF. Available at https://www.iso.org/standard/26153.html.

[32] Alfred Aho, M Lam, R Sethi, J Ullman (2006) *Compilers: Principles, Techniques, and Tools* (Addison-Wesley).

[33] Donald Knuth (1968) Semantics of context-free languages. *Math. Systems Theory 2*, p 127–145. https://doi.org/10.1007/BF01692511

[34] Software Engineering Institute (2016) SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. Available at https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf.

[35] Software Engineering Institute (2016) SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. Available at https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf.

[36] Open Web Application Security Project (2010) OWASP Secure Coding Practices-Quick Reference Guide. Available at https://owasp.org/www-project-secure-coding-practices-quick-reference-guide.

[37] GitHub (2008) GitHub. Available at https://github.com.

[38] FIRST (2021-2024) Exploit Prediction Scoring System (EPSS). Available at https://www.first.org/epss.

[39] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCWE Tool. Available at https://usnistgov.github.io/BF/info/tools/bfcwe-tool.

[40] Irena Bojanova, J J Guerrerio (Sept.-Oct. 2023) Labeling Software Security Vulnerabilities. *IEEE IT Professional*, Vol. 25, 5, pp 64–70. https://doi.org/10.1109/MITP.2023.3314368

[41] Peter Mell, Irena Bojanova (2024) NIST IR 8517, Hardware Security Failure Scenarios: Potential Weaknesses in Hardware Design. Available at https://doi.org/10.6028/NIST.IR.8517.

[42] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCVE Tool. Available at https://usnistgov.github.io/BF/info/tools/bfcve-tool.

[43] Chen Y, at all (2023) DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. Available at https://github.com/wagner-group/diversevul.

[44] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BF Tool. Available at https://usnistgov.github.io/BF/info/tools/bf-tool.

[45] Carlos EC Galhardo, P Mell, I Bojanova, A Gueye (December 2020) Measurements of the Most Significant Software Security Weaknesses. *2020 Annual Computer Security Applications Conference (ACSAC)*, p 154–164. https://doi.org/10.1145/3427228.3427257

[46] MITRE (2023) Weaknesses for Simplified Mapping of Published Vulnerabilities. Available at https://cwe.mitre.org/data/definitions/1003.html.

[46] MITRE (2023) Weaknesses for Simplified Mapping of Published Vulnerabilities. Available at https://cwe.mitre.org/data/definitions/1003.html.

[47] Kevin Poireault (Mar. 28, 2024) NIST Unveils New Consortium to Operate National Vulnerability Database. Available at https://www.infosecurity-magazine.com/news/nist-unveils-new-nvd-consortium/#:~:text=NVD's%20One%2Dto%2DFive%20Year,years%2C%20especially%20around%20software%20identification.

[48] Peter Mell, I Bojanova, Carlos EC Galhardo (May-Jun. 2024) Measuring the Exploitation of Weaknesses in the Wild. *IEEE IT Professional*, Vol. 26, 2, pp 14–21. https://doi.org/10.1109/MITP.2024.3399485

[49] NIST Common Platform Enumeration (CPE). Available at https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe.