Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight

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

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

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

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

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

I. Introduction

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

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

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

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

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

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

Disclaimer: Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor that they are necessarily the best available for the purpose.

jection, and the DVR bug in CVE-2014-0160 Heartbleed.

The rest of the paper is organized as follows: In Section II, we recall BF's approach and methodology. In Section III, we analyze the types of data check bugs and define the BF Data Check Bugs model. In Section IV, we present our new BF DVL and BF DVR classes. In Section V, we analyze the correspondence of the input, output, and injection-related CWEs to the new BF classes. In Section VI, we use the BF Data Check Bugs taxonomy to provide better, structured descriptions of real-world vulnerabilities (CVE entries [6]). Finally, in Section VII we discuss related works and in Section VIII we summarize the paper contributions and we propose future works.

II. BF Approach and Methodology

We use the terms software bug, weakness, and vulnerability as they are defined by Bojanova and Galhardo at [8]. We utilize the latest BF approach and methodology, as described in [9] and reiterate the main ideas here for better content flow.

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

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

Operations or operands improperness defines the causes. A consequence is the result of an operation over its operands. It becomes a final error or the cause for a next weakness.

A BF class is a taxonomic category of a weakness type, defined by a set of operations, all valid cause→consequence relations, and a set of attributes. The taxonomy of a particular bug or weakness is based on one BF class. Its description is an instance of a taxonomic BF class with one cause, one operation, one consequence, and their attributes. The operation binds the cause→consequence relation – e.g., under-restrictive sanitization policy leads to a final error known as SQL (Structured Query Language) injection.

CWEs coverage by any newly developed BF classes can be visualized via digraphs, based on CWEs parent-child relationships. Once analyzed, these digraphs can help understand CWEs structure and how they translate to BF.

The taxonomies of newly developed BF classes can be demonstrated by providing structured BF descriptions of appropriate CVEs.

The methodology for developing a BF class comprises identifying/defining: (1) the phase specific for a kind of bugs; (2) the operations for that phase; (3) the BF Bugs model with operations flow; (4) all causes; (5) all consequences that propagate as a cause to a next weakness; (6) all consequences that are final errors; (7) attributes useful to describe such a bug/weakness; (8) the possible sites in code; (9) CWE digraphs by class and consequence; (10) CVE test cases.

III. Data Check Bugs Model

Data should be checked to ensure proper results from its processing. It should adhere to object data types acceptable by the software. In [9], Bojanova and Galhardo, define an object as a piece of memory with well-defined size that is used to store primitive data or a data structure. As input, primitive data are checked and sanitized on string-of-characters level. A character – e.g., an ASCII (American Standard Code for Information Interchange) character – is a single symbol, such as an alphabet letter, a numeric digit, or a punctuation mark. Data structures in turn are built on primitive data but can also have a higher level of syntax and semantics rules.

Data check comprises data validation, where data syntax gets checked for being well-formed, and data verification, where data semantics gets verified for being accurate. The BF model separates data semantics check as data verification, although OWASP (Open Web Application Security Project) puts it under input validation [10].

Validation is about accepting or rejecting data based on its syntax: it checks if data are in proper format (grammar). For example, checking if data consist of digits only or checking the syntax of an XML (Extensible Markup Language) document against a DTD (XML Document Type Definition) is data validation. Once data syntax is checked it may be sanitized. Sanitization is about neutralizing, filtering, or repairing data via escaping, removing, or adding symbols in data, correspondingly. An example of data sanitization would be removing a special character such as '/' or adding a closing parenthesis ')'. The validate and sanitize operations use specific policies, such as to check against safelist, denylist, format (e.g., defined via regular expressions), or length. A safelist defines a set of known good content. A denylist defines a set of known bad content; it helps reject outright maliciously malformed data. Regular expressions define format patterns that data (viewed as strings) should match. They could be used for safelists and denylists.

Verification is about accepting or rejecting data based on its semantics: it checks if data have proper value (meaning). For example, checking if a start date is before an end date, or checking the type of an XML document against a PowerPoint XSD (XML Schema Definition) is data verification. Once data semantics is checked, it may be corrected via assigning a new value or via removing data. An example of data correction would be setting the size to the buffer's upper bound when the size that is supplied is larger than that upper bound. The verify and correct operations use specific policies to, for example, check data against a value (incl. NULL or list of values), quantity, range, type, or other business rules.

Data check bugs could be introduced at any of the *data validation* and *data verification* phases. Each bug or weakness involves one data check operation: validate, sanitize, verify, or correct. Each operation is over already-canonicalized[1] data and the policy (the rules) against which it gets checked.

[1]Canonicalization [11] operations, such as decrypt, cryptographic verify, decompress, or decode into format appropriate for the software, are performed before input check; the opposite operations are performed after output check. All these operations are not part of the BF Data Check Bugs classes.

The BF Data Check Bugs model (Fig. 1) helped us identify the phases and the operations where such bugs could occur. The phases correspond to the BF Data Check Bugs classes: Data Validation Bugs (DVL) and Data Verification Bugs (DVR). All data check operations are grouped by phase.

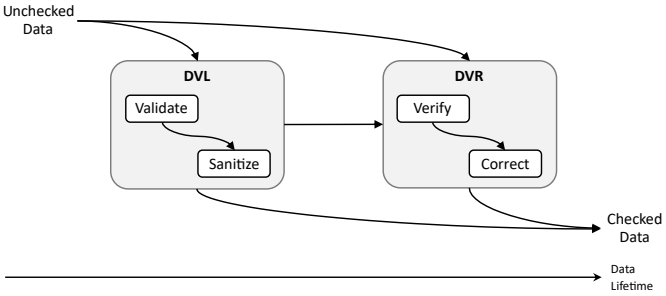


Fig. 1: The BF Data Check Bugs model. Comprises phases, corresponding to the BF classes DVL and DVR. Shows the data check operations flow.

The operations under DVL (Fig. 1) are on checking data syntax: *Validate* and *Sanitize*. See definitions of DVL operations in Table Ia. The operations under DVR (Fig. 1) are on checking data semantics: *Verify* and *Correct*. See definitions of DVR operations in Table Ib.

The possible flow between operations from different phases is depicted on Fig. 1 with arrows. Data could be validated and verified or only directly verified. The presented operations flow helps in identifying possible chains of bugs/weaknesses.

IV. BF Data Check Bugs Classes

We define the BF Data Check Bugs classes as follows:

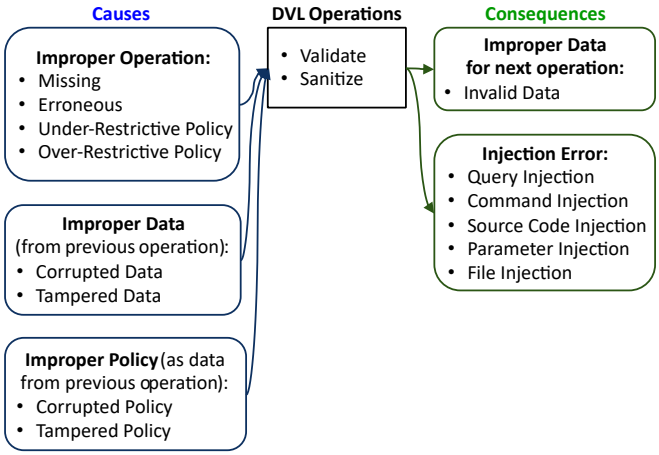
- Data Validation Bugs (DVL) – *Data are validated (syntax check) or sanitized (escape, filter, repair) improperly.*
- Data Verification Bugs (DVR) – *Data are verified (semantics check) or corrected (assign value, remove) improperly.*

Each of these classes represents a phase, aligned with the Data Check Bugs model (Fig. 1), and is comprised of sets of operations, cause→consequence relations, and attributes, allowing precise causal descriptions of data validation and data verification bugs/weaknesses.

Fig. 2 and Fig. 3 show the specific sets for data validation and data verification bugs, respectively. Only the values listed on the corresponding figure should be used to describe that kind of bugs or weaknesses.

A. Operations

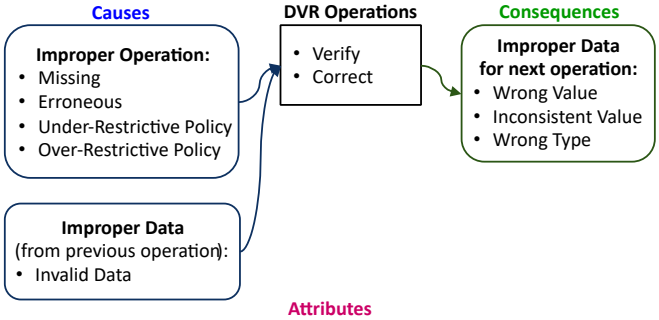
All BF classes are being designed to be orthogonal; their sets of operations should not overlap. The operations in which data check bugs could happen correspond to the operations in the BF Data Check Bugs model (Fig. 1) – as a reminder, the definitions are in Table I. The DVL operations are Validate and Sanitize. They reflect the improper check and modification of data syntax. The DVR operations are Verify and Correct. They reflect the improper check and modification of data semantics.



Attributes

Operation			Data
Mechanism:	**Source Code:**	**Execution Space:**	**State:**
• Safelist	• Codebase	• Local	• Entered
• Denylist	• Third Party	• Admin	• Stored
• Format	• Standard Library	• Bare-Metal	• In Use
• Length	• Processor		• Transferred

Fig. 2: The Data Validation (DVL) class.



Attributes

Operation			Data
Mechanism:	**Source Code:**	**Execution Space:**	**State:**
• Value	• Codebase	• Local	• Entered
• Quantity	• Third Party	• Admin	• Stored
• Range	• Standard Library	• Bare-Metal	• In Use
• Type	• Processor		• Transferred
• Other Rules			

Fig. 3: The Data Verification (DVR) class.

B. Causes

A cause is either an improper operation or an improper operand. If a BF class instance is the first in a chain describing a vulnerability, it is always caused by an improper operation. The values for improper data check operations are Missing, Erroneous, Under-Restrictive Policy, and Over-Restrictive Policy. See definitions and examples in Table II.

The operands of a data check operation are data and policy. See definitions in Table III. An improper operand is data or policy that has been modified by an operation of a previous weakness, such as decode, decrypt, and convert [8].

All values for an improper operand of a data check operation are defined in Table IV. Improper Policy as data from a previous weakness is a possible cause only for DVL. Improper Data could be of a primitive data type (e.g., a string, a number) or a data structure. Comments could be used to provide more details such as data type and variable name.

TABLE I: Operations

(a) DVL (Data Validation)

Operation Value	Definition
Validate	Check data syntax (proper form/grammar) in order to accept (and possibly sanitize) or reject it. Includes checking for missing symbols/elements.
Sanitize	Modify data (neutralize/escape, filter/remove, repair/add symbols) in order to make it valid (well-formed).

(b) DVR (Data Verification)

Operation Value	Definition
Verify	Check data semantics (proper value/meaning) in order to accept (and possibly correct) or reject it.
Correct	Modify data (assign new value, remove), in order to make it accurate.

TABLE II: Improper Operations

Value	Definition	Example
Missing	The operation is absent.	Missing data sanitization.
Erroneous	There's a bug in the operation implementation (incl. how it checks against a policy).	• Using not equal to (!=) when comparing to safelist values. • Using greater than (>) when checking for upper range.
Under-Restrictive Policy	Accepts bad data.	• Permissive safelist or regular expression. • Incomplete denylist.
Over-Restrictive Policy	Rejects good data.	Over-restrictive denylist or regular expression.

C. Consequences

A consequence is either a final error or a wrong result from the operation that propagates as an improper operand for a next weakness.

Improper validation or sanitization may directly lead to final injection errors. Any other improper data check in any of the phases (Fig. 1) would lead to improper data for an operation in a next weakness.

Improper Data is the only possible improper operand as a consequence from DVL or DVR. All its possible values are defined in Table V. As a consequence, improper data would become a cause for an operation of a next weakness. These consequence–cause transitions explain why Invalid Data appears in both Table IV and Table V.

The only kind of DVR consequences (Table Vb) is improper operand for next weakness. This means a DVR bug or weakness is always followed by another weakness or a chain of weaknesses leading to a final error such as buffer overflow or memory overflow.

All DVL final errors are injection errors. We also identify and define in Table VI five types of injection errors: query injection, command injection, source code injection, parameter injection, and file injection. All of them, except some source code injections, are due to data with missing, additional, or inconsistent special elements (symbols that can be interpreted as control elements or syntactic markers). The BF model separates query injection and command injection from source code injection, although they are commonly discussed under the umbrella term "code injection" (e.g., [3], [4], [12]). All the possible types of data check errors that end in failure right after the current bug/weakness (as an instance of a BF class) are DVL final errors, representing the types of injection errors.

All possible consequences for data check bugs are defined in Table V and Table VI. However, refer Fig. 2 and Fig. 3 for consequences applicable to each class.

TABLE III: Operands

Concept	Definition
Data	A string of characters (symbols like letter, digit, punctuation) with clearly defined syntax and semantics.
Policy	Lists or rules for checking data syntax and semantics. For example, safelist, denylist, format (e.g., DTD-XML Document Type Definition), and length; or value (incl. NULL or list of values), quantity, range, and type (e.g., a PowerPoint XSD).

TABLE IV: Improper Operands – as DVL/DVR Causes

(a) Improper Data (from previous operation) – as DVL Cause

Value	Definition
Corrupted Data	Unintentionally modified data due to a previous weakness (e.g., with a decompress or a decrypt operation) that if not sanitized would end-up as invalid data for next weakness.
Tampered Data	Maliciously modified data due to a previous weakness (e.g., with a deserialize, authorize, or crypto verify operation) that would lead to injection error.

(b) Improper Data (from previous operation) – as DVR Cause

Value	Definition
Invalid Data	Data with harmed syntax due to sanitization errors.

(c) Improper Policy (from previous operation) – as DVL Cause

Value	Definition
Corrupted Policy	Unintentionally modified policy due to a previous weakness (e.g., with a decompress operation).
Tampered Policy	Maliciously modified policy due to a previous weakness (e.g., with an authorize operation).

TABLE V: Improper Operands – as DVL/DVR Consequences

(a) Improper Data for Next Operation – as DVL/DVR Consequence

Value	Definition
Invalid Data	Data with harmed syntax due to sanitization errors.

(b) Improper Data for Next Operation – as DVR Consequence

Value	Definition
Wrong Value	Data with harmed semantics; not accurate value (e.g., outside of a range).
Inconsistent Value	Data value is inconsistent with the value of a related data (e.g., inconstancy between the value of a size variable and the actual buffer size).
Wrong Type	Data with wrong data type.

D. Attributes

An attribute provides additional useful information about the operation or its operands. All possible attributes for data check bugs are defined in Table VII. The operation attributes Source Code and Execution Space explain where a bug is – where the operation is in the program and where its code runs. The data attribute State explains where the data come from. See definitions of these attributes' values in Table VIIa.

Both DVL and DVR also have the operation attribute Mechanism but with different possible values that are specific

TABLE VI: Injection Errors – as DVL Consequences

Value	Definition	Example
Query Injection	Malicious insertion of condition parts (e.g., `or 1==1`) or entire commands (e.g., `drop table`) into an input used to construct a database query.	• SQL Injection • No SQL Injection • XPath Injection • XQuery Injection • LDAP Injection
Command Injection	Malicious insertion of new commands into the input to a command that is sent to an operating system (OS) or to a server.	• OS Command Injection • Regular Expression Injection • IMAP/SMTP Command Injection • Object Injection (JSON server side)
Source Code Injection	Malicious insertion of new code (incl. with <> elements) into input used as part of an executing application code.	• Cross Site Scripting (XSS) • CSS Injection • Eval Injection • EL Injection • JSON Injection (Client or Server Side)
Parameter Injection	Malicious insertion of data (e.g., with & parameter separator) into input used as parameter/argument in other parts of code.	• Argument Injection • Format String Injection • Email Injection • HTTP Header Injection (incl. Server Header Injection) • Reflection Injection • Flash Injection • CRLF Injection (incl. HTTP Header Splitting)
File Injection	Malicious insertion of data (e.g., with .. and / or with file entries) into input used to access/modify files or as file content.	• CSV, Temp, etc. File Injection • Log Entry Injection • XML Injection • CRLF Injection (incl. in as in log entry files) • Relative Path Traversal • Absolute Path Traversal

to the policies the operations could use to check data. See definitions of this attribute values in Table VIIb and Table VIIc.

E. Sites

A site for input/output check bugs is any part of the code that should check and sanitize data syntax or check and correct data semantics.

V. BF Data Check Bugs Taxonomy as CWE Extension

In this section, we analyze the correspondence of the input, output, and injection-related CWEs [5] to the two new BF DVL and DVR classes. We show that the BF classes cover all related CWEs, and potentially beyond, while (as demonstrated later in Section VI) providing a better structured way for describing these kinds of bugs/weaknesses.

The BF classes ensure precise causal descriptions, as a weakness is described via one cause, one operation, and one consequence, while the CWEs only enumerate weaknesses. The CWEs exhaustive list approach is prone to gaps in coverage: some weakness types may be missing. The CWEs also have overlaps in coverage, including via over detailing (e.g., CWE-23 children's path traversal variations). While by their nature, the BF classes are complete and orthogonal, assuring no gaps and no overlaps in coverage. We map a CWE

TABLE VII: Attributes

(a) DVL and DVR Attributes

Name	Value	Definition
Source Code	Codebase	The operation is in the programmer's code – in the application itself.
	Third Party	The operation is in a third-party library.
	Standard Library	The operation is in the standard library for a particular programming language.
	Language Processor	The operation is in the tool that allows execution or creates executables (compiler, assembler, interpreter).
Execution Space	Local	The bugged code runs in an environment with access control policy with limited (local user) permission.
	Admin	The bugged code runs in an environment with access control policy with unlimited (admin user) permission.
	Bare-Metal	The bugged code runs in an environment without privilege control. Usually, the program is the only software running and has total access to the hardware.
State	Entered	Data comes from user interface (e.g., text field).
	Stored	Data comes from permanent storage (e.g., file, database on a storage device).
	In Use	Data comes from volatile storage (e.g., RAM, cache memory).
	Transferred	Data comes via network (e.g., connecting analog device or another computer).

(b) DVL Attribute

Name	Value	Definition
Mechanism	Safelist	Policy based on a set of known good content.
	Denylist	Policy based on a set of known bad content; helps reject outright maliciously malformed data.
	Format	Policy based on syntax format (e.g., defined via regular expression).
	Length	Policy based on allowed number of characters in data. Note that this is not about the data value as size of an object.

(c) DVR Attribute

Name	Value	Definition
Mechanism	Value	Check data for a specific value (incl. NULL or list of values).
	Quantity	Check data for a specific measurable value (e.g., size, time, rate, frequency).
	Range	Check data are within a (min, max) interval.
	Type	Check data type (e.g., int, float, XSD-XML Schema Definition, or specific structure/object).
	Other Rules	Check data against other business logic.

to a BF class by an operation and/or a consequence from the lists defining the BF class. Through these relationships, the BF classes can be viewed as structured extensions to the input, output, and injection-related CWEs.

The BF Data Check Bugs classes relate to particular CWEs by BF DVL and DVR operations and/or consequences. We generated a digraph of all input- and output-check-related CWEs, including the injection-related CWEs, to show this correspondence both by operation (Fig. 4) and by consequence (Fig. 5). In the digraph, an edge starts at a parent CWE and ends at a child CWE. The outline style of a CWE node indicates the CWE level of abstraction: pillar, class, base, or variant.

Almost all of the 162 CWEs, visualized on the digraph, can be tracked as descendants of the pillar CWE-707; excep-

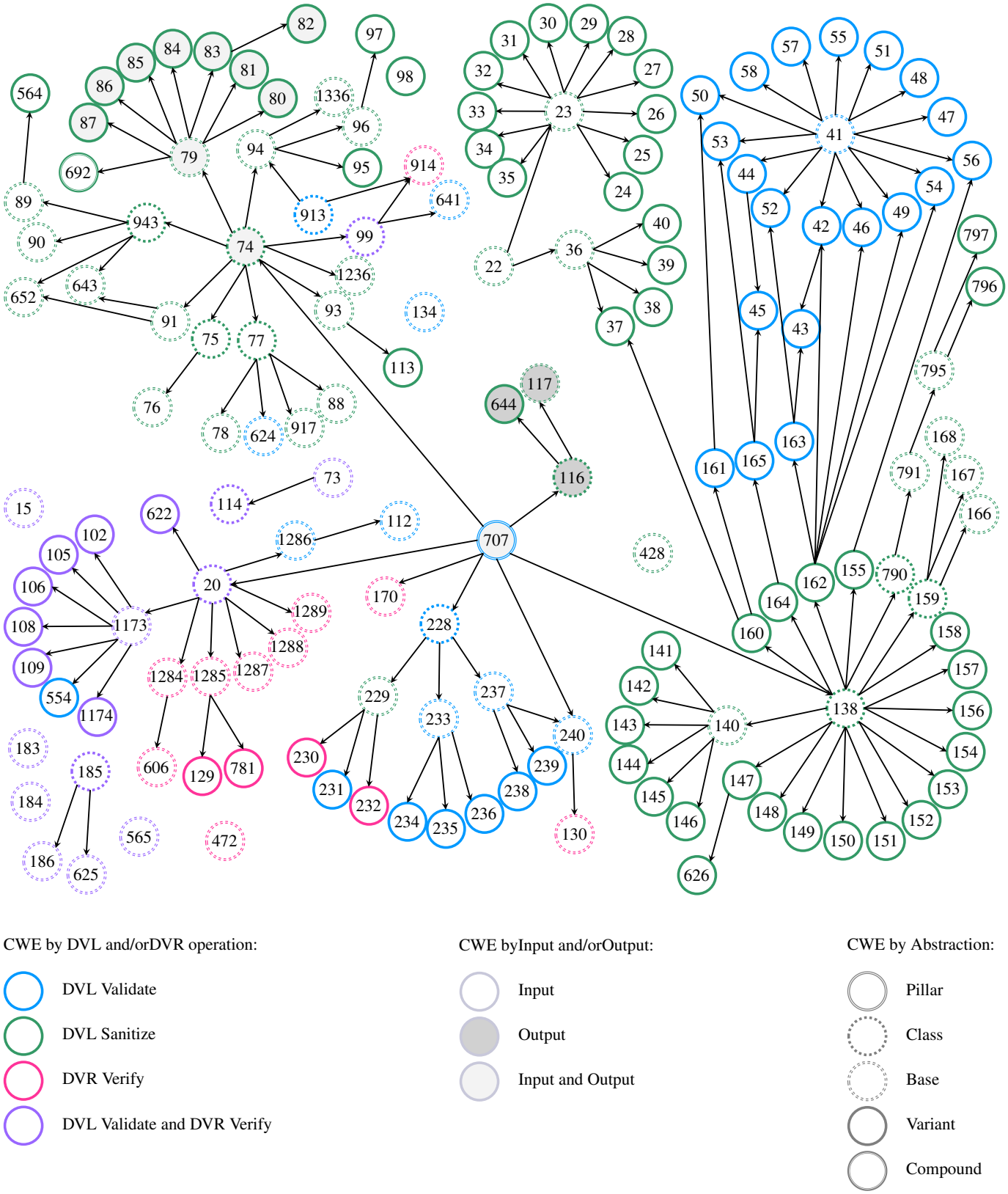


Fig. 4: A digraph of the input- and output-check-related CWEs (including injection-related CWEs), *mapped by BF DVL and BF DVR operations*. Each node represents a CWE by its identifier (ID). Each arrow represent a parent-child relationship. → Click on an ID to open the CWE entry.

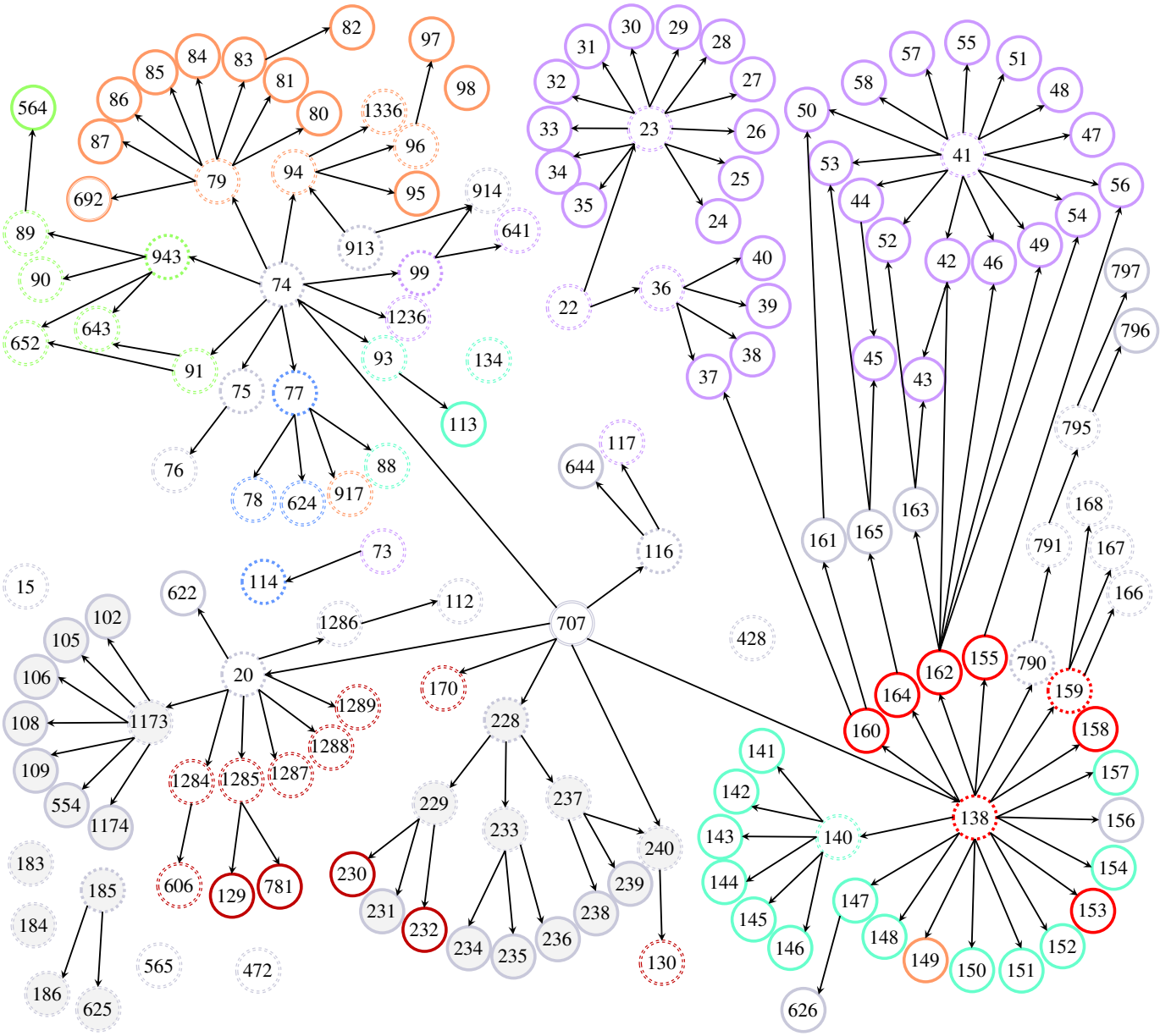


Fig. 5: A digraph of the input- and output-check-related CWEs (including the injection-related CWEs), *mapped by BF DVL and BF DVR consequences*. Each node represents a CWE by its identifier (ID). Each arrow represent a parent-child relationship. → Click on an ID to open the CWE entry.

tions are CWEs 15, 73 (its child 114), 98, 134, 183, 184, 185 (and its children 186 and 625), 428, 472, 565, and 913.

Analyzing the digraph, we see that the pillar CWE-707 reflects neutralization of malformed messages; it is quite broad. It is the parent of several CWEs that are not strictly input/output check related.

The digraph helped us identify seven distinct clusters of CWEs with parent CWEs 20, 22, 41, 74, 116, 138, and 228.

Most CWEs are about input check. CWEs-116 and its children, CWEs 117 and 644, are about output check. CWEs 80, 81, 82, 83, 84, 85, 86, and 87 are about both input check and output check.

In Fig. 4, the outline color of a CWE node indicates the BF class(es) and operation(s) associated with that CWE: DVL Validate, DVL Sanitize, DVR Verify, and DVL Validate and DVR Verify. There are no CWEs that relate to DVR Correct. The CWE-20 cluster is the only one that corresponds to DVR Verify. However, some of its descendants correspond also to DVL Validate. The CWE-41 and CWE-228 (except 3 nodes) clusters correspond to DVL Validate. The rest of the clusters, CWE-22, CWE-74, CWE-116, and CWE-138, correspond to DVL Sanitize.

The digraph on Fig. 5 illustrates how the CWEs map to BF DVL and DVR by consequence (Table V and Table VI).

In Fig. 5, the outline color of a CWE node indicates the BF class consequence associated with that CWE: Query Injection, Command Injection, Source Code Injection, Parameter Injection, File Injection, Invalid Data, Wrong Value, Inconsistent Value, or Wrong Type.

The CWE-74 cluster is all about injection; the largest group there is Source Code Injection, then are Query Injection and Command Injection, and a few nodes of File Injection and Parameter Injection. CWEs 74, 75, 76, 913, 914 are not colored, as they are about general injection errors. The CWE-22 and CWE-41 clusters are all about File Injection. The CWE-138 cluster is mostly about Parameter Injection and some Source Code Injection. Some of the CWE-138 nodes correspond to the DVL Invalid Data for Next Operation consequence. The CWE-20 cluster (which is DVR related) has nodes corresponding to the DVR Wrong Data for Next Operation consequences; however, some of these CWEs describe only causes and do not list consequences.

All relationships to input, output, and injection CWEs (Fig. 5) by BF DVL and BF DVR consequences are as follows:

1) The DVL class relates to CWEs through its Wrong Data for Next Operation consequence as follows:
   - Invalid Data – CWEs: 138, 153, 155, 158, 159, 160, 162, and 164.
2) The DVL class relates to CWEs also through its Injection Error consequences as follows [2]:
   - Query Injection – CWEs: 89, 90, 91, 564, 643, 652, and 943;
   - Command Injection – CWEs: 77, 78, and 624;
   - Source Code Injection – CWEs: 79, 80, 81, 83, 82, 85, 86, 87, 94, 95, 96, 97, 98, 149, 692, 917, and 1336;
   - Parameter Injection – CWEs: 88, 93, 113, 134, 140, 141, 142, 143, 144, 145, 146, 147, 148, 150, 151, 152, 154, and 157;
   - File Injection – CWEs: 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 73, 99, 117, 641, and 1236.
3) The DVR class relates to CWEs through its Wrong Data for Next Operation consequence as follows:
   - Wrong Value – CWEs: 129, 170, 606, 781, 1285, and 1289;
   - Inconsistent Value – CWEs: 130, 230, 232, and 1288;
   - Wrong Type – CWEs: 1284, and 1287.
4) The following CWEs reflect only DVL or DVR causes: 102, 105, 106, 108, 109, 183, 184, 185, 186, 228, 229, 231, 233, 234, 235, 236, 237, 238, 239, 240, 554, 625, 1173, and 1174.

The BF Data Check Bugs classes present a taxonomy with structured cause→consequence relations that is complete and orthogonal. It could be viewed as a structured extension over the CWEs related to Injection Errors, Invalid Data, Wrong Value, Inconsistent Value, and Wrong Type (Fig. 5). It is a taxonomy that explains the causal relationships between weaknesses and would be easier to use than the nested hierarchical CWEs.

Many bug reporting tools use the CWE [5] to describe found bugs/weaknesses [13]. As a structured extension over the input, output, and injection CWEs, the BF Data Check taxonomy can be used to report found data check bugs/weaknesses (including those leading to injection errors). Fig. 4 shows how input/output and injection CWEs translate to BF DVL and BF DVR by operation; Fig. 5 shows how they translate by consequences.

VI. Showcases

In this section, we use the new BF Data Check Bugs classes for precise descriptions of real-world software vulnerabilities. We also provide the real-world fixes of each bug.

A. CVE-2020-5902 – BIG-IP TMUI RCE

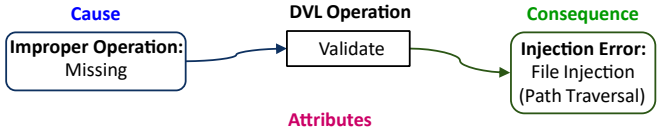
This vulnerability is listed in CVE-2020-5902. It was one of the top routinely exploited vulnerabilities in 2020 [14]. It was identified by Mikhail Klyuchnikov [15].

1) Brief Description: BIG-IP is a family of server-side products from F5 Inc. focused on availability, performance, and security. In several versions, its Traffic Management Interface (TMUI), known as the BIG-IP Configuration utility, allows Remote Code Execution (RCE).

2) Analysis: TMUI of BIG-IP accepts `/..;/` via the login interface `/tmui/login.jsp`. However, the Apache Tomcat treats `/..;/` as `/../`, which is a relative path for going one directory up. This allows a malicious user to bypass authentication, save and open files, and run arbitrary commands on the host. Fig. 6 presents the BF taxonomy for this vulnerability.

3) The Fix: To fix the bug, input validation (e.g., via the "`.*\.\.;.*`" regular expression) should be added to reject any `/..;/` elements [16].

[2]Some injection-related CWEs are not too specific and may end up as Invalid Data for an operation of a next weakness.



Attributes

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

Fig. 6: BF for CVE-2020-5902 – BIG-IP TMUI RCE

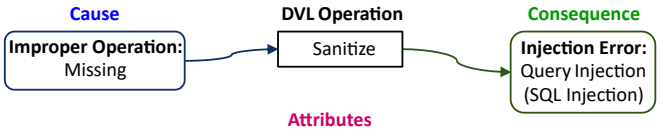
B. CVE-2019-10748 – Sequelize SQL Injection

This vulnerability is listed in CVE-2019-10748. It was discovered by the Snyk Security Team [17]. The source code could be found at [18].

1) Brief Description: Sequelize is an Object-Relational Mapper for `Node.js`. It supports Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server; it facilitates transaction support, relations, and lazy loading [19]. In several versions `query-generator.js` allows SQL injection.

2) Analysis: User input path is not sanitized for MySQL/MariaDB syntax in a JSON (JavaScript Object Notation) object. Fig. 7 presents the BF taxonomy for this vulnerability.

3) The Fix: To fix the bug, the developers check the input paths syntax and sanitize it.



Attributes

Operation			Data
Mechanism: • Format	**Source Code:** • Codebase (`query-generator.js`)	**Execution Space:** • Local	**State:** • Transferred (via network)

Fig. 7: BF for CVE-2019-10748 – Sequelize SQL Injection

C. CVE-2014-0160 – Heartbleed Buffer Overflow

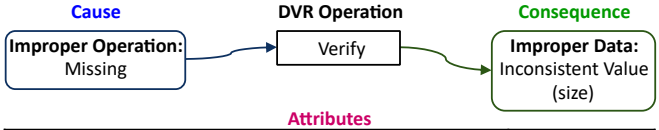
This vulnerability is listed in CVE-2014-0160 and discussed in [20]. The source code could be found at [21].

Heartbleed is partially described in [9] using the BF MAD (Memory Addressing Bugs) and the BF MUS (Memory Use Bugs) classes. Here we complete the BF taxonomy for Heartbleed by describing the DVR bug that starts the chain of weaknesses leading to buffer overflow.

1) Brief Description: Heartbleed is a vulnerability due to a bug in the OpenSSL that allows a server to read over the bounds of a buffer. Using the heartbeat extension tests in TLS (Transport Layer Security) and DTLS (Datagram Transport Layer Security) protocols, a user can send a heartbeat request to a server. The request contains a string and a `payload` unsigned integer, which value is expected to be the string size but could be as big as `65535`.

2) Analysis: Detailed analysis is provided in [9]. Fig. 8 presents the BF taxonomy for the Heartbeat DVR bug.

3) The Fix: To fix the bug, the OpenSSL team added a value verification for the array size [22].



Attributes

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

Fig. 8: BF DRV bug from DVR–MAD–(MUS & MUS) chain of CVE-2014-0160 – Heartbleed Buffer Overflow in [9]

VII. Related Works

In this work, we introduce BF's new classes for data validation and data verification bugs. They can be used to describe input data check bugs that lead to injection errors or to improper (e.g., inconsistent) data that would cause other software errors (e.g., buffer overflow).

Several authors attempted to create successful taxonomies of bugs/weaknesses that lead to security failures. Hui et al reviewed those taxonomies in [23]. Data validation (usually called parameter or input validation) is a common category across the different taxonomies reviewed by them. The new BF Data Check Bugs taxonomy differs from any of these taxonomies as it allows describing how a security failure emerges from a bug by a chain of weaknesses. For a bug to exist, there should be a particular cause leading to a particular consequence. In BF, the kinds of causes relate either to improper operations or to improper operands. The cause of one weakness could be the consequence of a previous weakness. This chain of weaknesses eventually ends in a software error that leads to a security failure. This approach clearly explains, for example, that the well-known Heartbleed vulnerability starts with a Data Verification Bug, which leads to memory-related weaknesses, ending in a buffer overflow error. Using any of the reviewed taxonomies, it would not be possible to describe and understand the interrelationship between weaknesses nor how the failure (e.g., exposure of sensitive information in Heartbleed) emerges.

Ray and Ligatti [12] formally define what they call code injection on output (CIAO). The reasoning behind their definition is that injection errors happen when untrusted input propagates unmodified to output. The CIAO definition is equivalent to the union of our definitions of query injection, command injection, and source code injection. All, except some source code injections, are related to unchecked symbols that propagate from input to output.

Ray and Ligatti also define code interference attacks on outputs (CIntAO). The reasoning behind their CAIntO definition is analogous to the reasoning for our Parameter Injection definition – maliciously inserted data causes an unexpected behavior that leads to a security failure.

In some sources (e.g., [4], [24], [25]), the term "code injection" is used to describe an RCE failure, caused for example, by buffer overflow. Although this kind of security failure is not caused by source code injection (as we have defined it), for some buffer overflow errors a data verification bug may be the first bug in the chain leading to that error (see Section VI-C). Using BF to describe such a vulnerability would help clearly separate source code injection from buffer overflow that leads to arbitrary code execution. This

exemplifies how BF can help avoid confusion in vulnerability descriptions and improve communication about bugs, weaknesses, and the security failures caused by them.

Currently, several institutions provide lists of security bugs/weaknesses [5], [26]–[28]. From these, we recognize the MITRE CWE as the most widely known and used one. We discuss in Section V how BF extends the CWE exhaustive list approach, as well as how to map CWEs to BF classes and extend the CWE based bug reports with BF descriptions.

VIII. Conclusion

In this paper, we introduce two new BF classes: Data Validation Bugs (DVL) and Data Verification Bugs (DVR). We present their operations, possible causes, consequences, attributes, and sites. We show how they cover all CWEs related to input/output validation, verification, as well as to injection. We analyze particular input data check and injection vulnerabilities and provide their precise BF descriptions. The BF structured taxonomies show the initial error in code (the bug), providing a quite concise and still far more clear description than the unstructured explanations in current repositories, advisories, and publications. The BF Data Check Bugs taxonomy can be used by bug reporting tools, as it is a structured extension over the input, output, and injection-related CWEs [5]. To our knowledge there is no other bug taxonomy that allows precise causal descriptions of data validation and data verification bugs/weaknesses.

Future work should identify and describe more CVEs related to data validation and data verification bugs/weaknesses, evaluating the BF Data Check Bugs taxonomy for usability. In such an evaluation, a machine learning algorithm or multiple analysts would classify and describe newly reported bugs [29], while helping improve BF's taxonomy by fine-tuning the classes.

The CWE digraphs by BF class consequences should be deeply analyzed. Generation of digraphs with CWEs related to particular software errors (e.g., injection errors), detecting corresponding clusters, and understanding their relationships would create a comprehensive view of the CWE model for researchers and practitioners. In turn, comparing and contrasting the CWE's exhaustive list of weaknesses with all the possible consequence-cause transitions to other BF classes would improve BF as a tool for describing CVEs.

We will continue developing orthogonal BF classes that cover and extend the CWE weakness types.

References

[1] C. E. Galhardo, P. Mell, I. Bojanova, and A. Gueye, "Measurements of the Most SignificantSoftware Security Weaknesses," in *Proc. ACSAC '20: Annual Computer Security Applications Conference*, 2020, pp. 154–164. DOI: 10.1145/3427228.3427257.

[2] A. Gueye, C. E. Galhardo, I. Bojanova, and P. Mell, "A Decade of Reoccurring Software Weaknesses," *IEEE Security & Privacy*, in press, 2021. DOI: 10.1109/MSEC.2021.3082757.

[3] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Conf. Record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, vol. 41, 2006, pp. 372–382. DOI: 10.1145/1111037.1111070.

[4] A. Francillon and C. Castelluccia, "Code Injection Attacks on Harvard-architecture Devices," in *Proc. 15th ACM Conf. Computer and Commun. Security*, 2008, pp. 15–26. DOI: 10.1145/1455770.1455775.

[5] MITRE, *Common Weakness Enumeration (CWE)*, Accessed: 2021-08-28, 2021. [Online]. Available: https://cwe.mitre.org.

[6] ——, *Common Vulnerabilities and Exposures (CVE)*, Accessed: 2021-08-28, 2020. [Online]. Available: https://cve.mitre.org/.

[7] NVD, *National Vulnerability Database (NVD)*, Accessed: 2021-08-28, 2021. [Online]. Available: https://nvd.nist.gov.

[8] *The Bugs Framework*, 2020. [Online]. Available: https://samate.nist.gov/BF/.

[9] I. Bojanova and C. E. Galhardo, "Classifying Memory Bugs Using Bugs Framework Approach," in *2021 IEEE 45nd Annu. Computer, Software, and Applications Conf. (COMPSAC)*, 2021, in press.

[10] CheatSheets Series Team, *Input Validation Cheat Sheet*, Accessed: 2021-08-28, 2017. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

[11] W. L. Fithen, *Ensure that Input Is Properly Canonicalized*, Accessed: 2021-08-28, 2013. [Online]. Available: https://us-cert.cisa.gov/bsi/articles/knowledge/coding-practices/ensure-input-properly-canonicalized.

[12] D. Ray and J. Ligatti, "Defining Code-injection Attacks," in *Proc. of the 39th Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, vol. 47, 2012, pp. 179–190. DOI: 10.1145/2103656.2103678.

[13] SAMATE Team, *Static Analysis Tool Exposition (SATE)*, Accessed: 2021-08-28, 2021. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate.

[14] CISA, ACSC, NCSC, and FBI, *Top Routinely Exploited Vulnerabilities*, Product ID:AA21-209A, 2021. [Online]. Available: https://us-cert.cisa.gov/sites/default/files/publications/AA21-209A_Joint_CSA%20Top%20Routinely%20Exploited%20Vulnerabilities.pdf.

[15] M. Klyuchnikov, *Remote Code Execution in F5 Big-IP*, Accessed: 2021-08-28, 2020. [Online]. Available: https://swarm.ptsecurity.com/rce-in-f5-big-ip/.

[16] RIFT: Research and Intelligence Fusion Team, *RIFT: F5 Networks K52145254: TMUI RCE vulnerability CVE-2020-5902 Intelligence*, Accessed: 2021-08-28, 2020. [Online]. Available: https://research.nccgroup.com/2020/07/05/rift-f5-networks-k52145254-tmui-rce-vulnerability-cve-2020-5902-intelligence/.

[17] Snyk Security Team, *SQL Injection*, Accessed: 2021-08-28, 2019. [Online]. Available: https://snyk.io/vuln/SNYK-JS-SEQUELIZE-450221.

[18] Sequelize Team, *Sequelize ORM*, Accessed: 2021-08-28, 2019. [Online]. Available: https://github.com/sequelize/sequelize/commit/a72a3f5.

[19] ——, *Sequelize ORM*, Accessed: 2021-08-28, 2019. [Online]. Available: https://sequelize.org/.

[20] Synopsys Inc, *The heartbleed bug*, Accessed: 2021-08-28, 2014. [Online]. Available: https://heartbleed.com/.

[21] Open SLL Team, *OpenSSL*, Accessed: 2021-08-28, 2014. [Online]. Available: https://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=ssl/d1_both.c;h=0a84f957118afa9804451add380eca4719a9765e;hb=4817504d069b4c5082161b02a22116ad75f822b1.

[22] OpenSSL Team, *OpenSSL*, Accessed: 2021-08-28, 2014. [Online]. Available: https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3.

[23] Z. Hui, S. Huang, Z. Ren, and Y. Yao, "Review of Software Security Defects Taxonomy," in *Int. Conf. Rough Sets and Knowledge Technology*, 2010, pp. 310–321.

[24] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proc. 2013 IEEE Symp. Security and Privacy*, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[25] D. Mitropoulos and D. Spinellis, "Fatal Injection: a Survey of Modern Code Injection Attack Countermeasures," *PeerJ Comput. Sci.*, vol. 3, e136, 2017. DOI: 10.7717/peerj-cs.136.

[26] OWASP, *OWASP Top Ten*, Accessed: 2021-08-28, 2017. [Online]. Available: https://owasp.org/www-project-top-ten/.

[27] NSA/CAS, "NSA/CAS Static Analysis Tool Study - Methodology," Center for Assured Software National Security Agency, Tech. Rep., 2011. [Online]. Available: https://www.nist.gov/system/files/documents/2021/03/24/CAS%5C%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.

[28] R. C. Seacord, *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley, 2014.

[29] T. M. Adhikari and Y. Wu, "Classifying software vulnerabilities by using the bugs framework," in *8th Inter. Symp. Digital Forensics and Security (ISDFS)*, 2020, pp. 1–6. DOI: 10.1109/ISDFS49300.2020.9116209.