Classifying Memory Bugs Using Bugs Framework Approach

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

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

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

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

I. Introduction

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

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

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

II. BF Approach and Methodology

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

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

BF describes a bug or a weakness as an improper state and its transition. The transition is to another weakness or to a failure. An improper state is defined by the tuple ($\texttt{operation}$, $\texttt{operand}_1$, $\cdots$, $\texttt{operand}_n$), 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. Rarely an intermediate state may also have a bug, which if fixed will also resolve the vulnerability. The final state, the failure, is caused by a final error (undefined or exploitable system behavior), which usually directly relates to a CWE [2]. A transition is the result of the operation over the operands.

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

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

We call a BF class the set of operations, the valid cause→consequence relations for these operations, their attributes, and sites. The attributes are qualifiers for the operations and the operands that help understand how severe a bug is. The sites show where in code a bug might occur. The BF classes are orthogonal by design; their sets of operations do not overlap.

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., deallocation via a dangling pointer leads to a final error known as double free [5].

The methodology for developing a BF class is as follows: (1) Identify the phase specific for a kind of bugs. (2) Identify the operations for that phase. (3) Define a BF bugs model showing operations flow. (4) Identify all causes. (5) Identify all consequences that propagate as a cause to a next weakness. (6) Identify all consequences that are final errors. (7) Identify attributes useful to describe such a bug/weakness. (8) Identify possible sites in code.

III. Memory Bugs Model

Each memory related bug or weakness involves one memory operation. Each *operation* is over a region of memory or over the address needed to reach it. That memory is used for storing data and has an important property: it is finite. It has *boundaries* and it has *size*. We call this piece of memory, with a well-defined size, an *object*. It is used to store a primitive data or a data structure. The memory address should be held by at least one *pointer* or determined as an offset on the stack, otherwise the object will be unreachable. The object and the pointer are the operands of that memory operation (see definitions in Table III).

Memory bugs could be introduced at any of the phases of an object's lifecycle: *address formation*, *allocation*, *use*, and *deallocation*. The BF Memory Bugs Model helps identify where in these phases bugs could occur (Fig. 1). The phases correspond to the BF memory bugs classes: Memory Addressing Bugs (MAD), Memory Allocation Bugs (MAL), Memory Use Bugs (MUS), and Memory Deallocation Bugs (MDL). All possible memory operations are grouped by phase. The presented operations flow helps in identifying possible chains of bugs/weaknesses.

The operations under MAD (Fig. 1) are on forming or modifying a pointer: *Initialize*, *Reposition*, and *Reassign*. Bugs in pointer initialization could result in pointers to meaningless objects. Moving a pointer via a bugged Reposition could get it pointing outside the object bounds. Bugs in Reassign could connect a pointer to a wrong object. See definitions of MAD operations in Table Ia.

The operations under MUS are on reading or writing the content of an object through one of its pointers: *Initialize*, *Read*, *Write*, *Clear*, and *Dereference*. Bugs in object initialization could lead to use of random or malicious data. Bugs in write could alter data wrongly. Bugs in Clear could leak confidential information such as passwords and cryptographic private keys. Bugs in Dereference are practically unsuccessful reading or unsuccessful writing. See definitions of MUS operations in Table Ic.

The operations under MAL are on creating an object or extending it through one of its pointers: *Allocate*, *Extend*, and *Reallocate–Extend* (see definitions in Table Ib). The

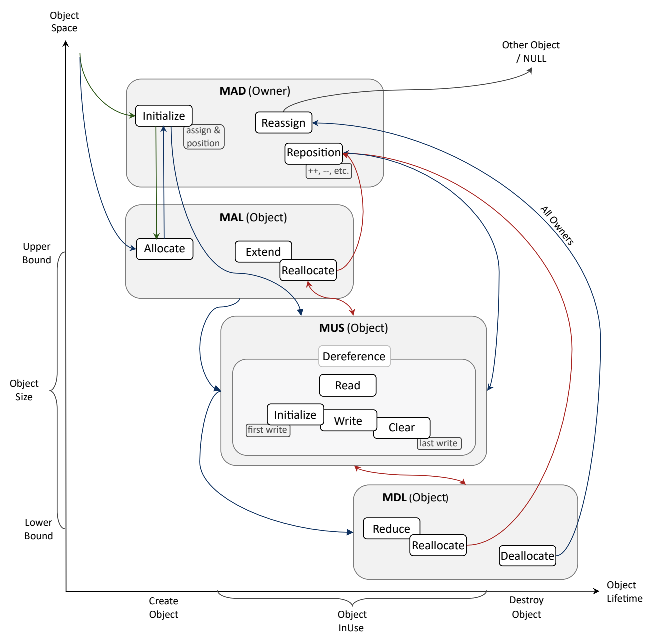


Fig. 1: The BF Memory Bugs Model. Comprises four phases, corresponding to the BF classes MAD, MAL, MUS, and MDL. Shows the memory operations flow: blue arrows – the main flow; green arrows – flow for allocation at a specific address; red – extra flow in case of reallocation.

operations under MDL are on destroying or reducing an object through one of its pointer: *Deallocate*, *Reduce*, and *Reallocate–Reduce* (see definitions in Table Id). Both MAL and MDL operations affect the boundaries and the size of the object. Bugs in Reallocate may concern multiple pointers to the same object. Allocation in excess or failure to deallocate unused objects could exhaust memory. Excessive reduction of allocated memory could lead to an object that is too little for the data it needs to store.

The possible flow between operations from different phases is depicted on Fig. 1 with colored arrows: blue is for the main flow; green is for allocation requested at a specific address; red is for the extra flow in case of reallocation.

Following the blue arrows, the very first operation is MAL Allocate an object. Following the green arrows, the first operation is MAD Initialize a pointer. Next operation, following the blue arrows, should be MAD Initialize the pointer to the address returned by Allocate. While, following the green arrows, next operation should be MAL Allocate an object at the address the pointer holds.

After an object is allocated and its pointer is initialized, it can be used via MUS Read or MUS Write. The boundaries and the size of an object are set at allocation, then they can be changed by any MAL or MDL operation.

If an object is owned by more than one pointer, Reallocate (in MAL or MDL) should be followed by Reposition over all these owners. A Deallocate an object operation should properly be followed by Reassign of all its pointers to either `NULL` or another object.

IV. BF Memory Bugs Classes

We define the BF Memory Bugs classes as follows:

TABLE I: Operations

(a) MAD (Memory Addressing)

Operation Value	Definition
Initialize (pointer)	The first assign of an object address to a pointer; positions the pointer at the start of the object.
Reposition	Changes the pointer to another position inside its object.
Reassign	Changes the pointer to a different object.

(b) MAL (Memory Allocation)

Operation Value	Definition
Allocate	Reserves space in memory for an object; defines its initial boundaries and size.
Extend	Allocates additional memory for an object in the same space; redefines its boundaries and size.
Reallocate–Extend	Allocates a new larger piece of memory for an object at a new address, copies the object content there, reassigns its pointer, and deallocates the previous piece of memory.

(c) MUS (Memory Use)

Operation Value	Definition
Initialize (object)	The first write into an object, after it is allocated.
Read	Gets content from an object.
Write	Puts content into an object.
Clear	The very last write into an object, before it is deallocated.
Dereference	Overreaches Initialize, Read, Write, and Clear, focus is on object access, no matter if it's for reading or for writing.

(d) MDL (Memory Deallocation)

Operation Value	Definition
Deallocate	Releases the allocated memory of an object.
Reduce	Deallocates part of the object memory; redefines its boundaries and size.
Reallocate–Reduce	Allocates a new smaller space in memory for an object at a new address, copies part of the object content there, reassigns the pointer, and deallocates the previous piece of memory.

Memory Addressing Bugs (MAD) – *The pointer to an object is initialized, repositioned, or reassigned to an improper memory address.*

Memory Allocation Bugs (MAL) – *An object is allocated, extended, or reallocated (while extending) improperly.*

Memory Use Bugs (MUS) – *An object is initialized, read, written, or cleared improperly.*

Memory Deallocation Bugs (MDL) – *An object is deallocated, reduced, or reallocated (while reducing) improperly.*

Each of these classes represents a phase, aligned with the Memory Bugs Model, and is comprised of sets of operations, cause→consequence relations, and attributes. Fig. 2, Fig. 3, Fig. 4, and Fig. 5 show the specific sets for memory addressing, allocation, use, and deallocation 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 memory bugs could happen are defined in Table I.

The MAD operations are: Initialize (Pointer), Reassign, Reposition. They reflect improper formation of an address.

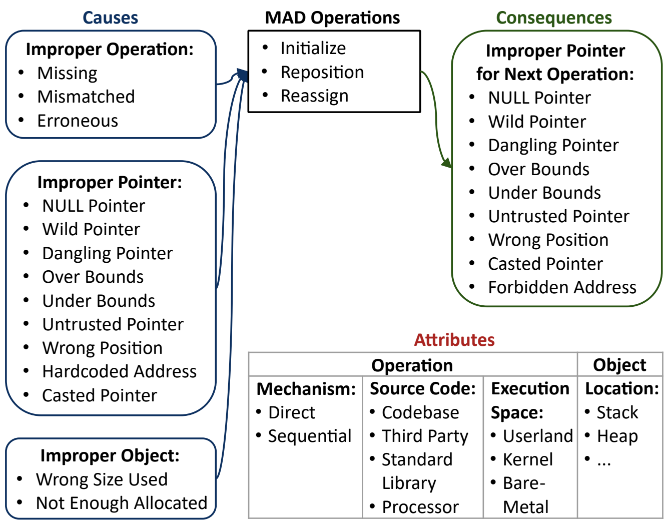


Fig. 2: The Memory Addressing Bugs (MAD) class.

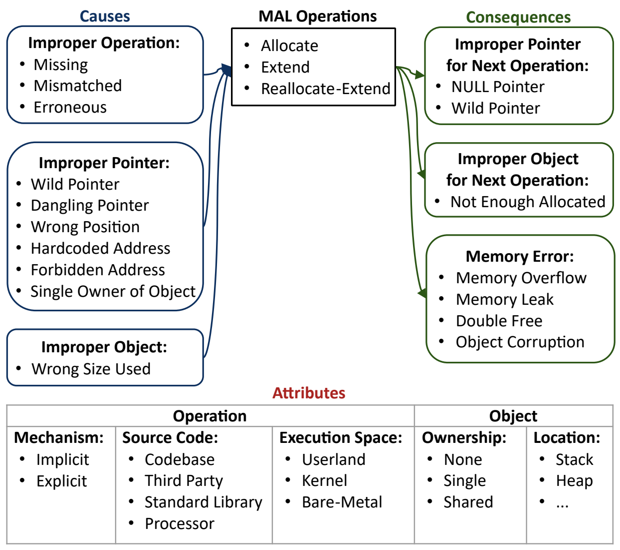


Fig. 3: The Memory Allocation Bugs (MAL) class.

The MAL operations are: Allocate, Extend, and Reallocate–Extend. They reflect improper formation of an object. The MUS operations are: Initialize (Object), Dereference, Read, Write, Clear. They reflect improper use of an object. The MDL operations are: Deallocate, Reduce, Reallocate–Reduce. They reflect improper release of an object. MAD Initialize and MUS Initialize are not overlapping, as the former is about the address, the latter is about the object.

B. Causes

A cause is either an improper operation or an improper operand. The values for improper memory operations are: *Missing*, *Mismatched*, and *Erroneous*. See definitions in Table II. The operands of a memory operation are pointer and object. See definitions in Table III. All values for improper operands of a memory operation are defined in Table IV.

An improper pointer could be a reference. Comments could be used to provide details, such as the pointer or reference identifier. An improper object could be a primitive

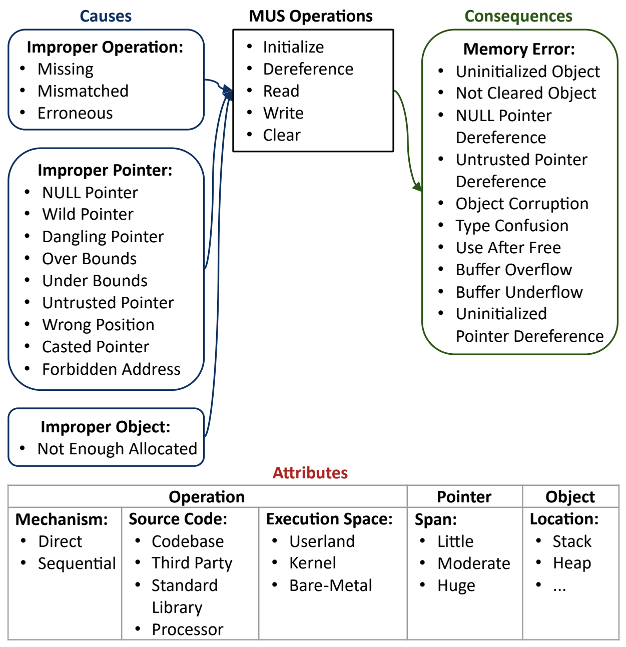


Fig. 4: The Memory Use Bugs (MUS) class.

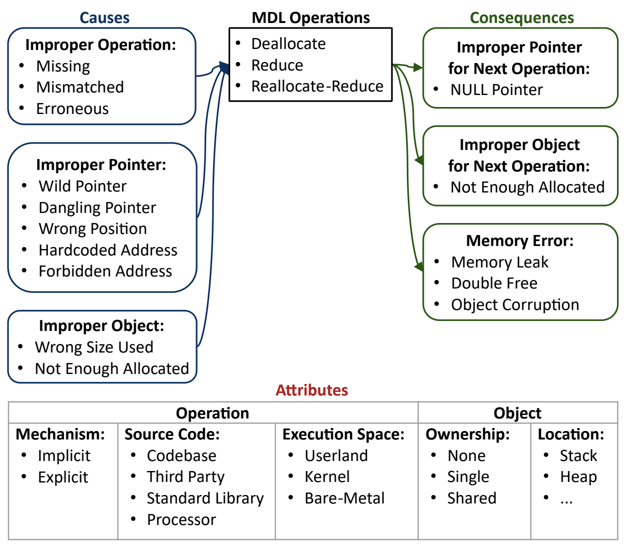


Fig. 5: The Memory Deallocation Bugs (MDL) class.

TABLE II: Improper Operations

Value	Definition	Example
Missing	The operation is absent.	Missing object initialization.
Mismatched	The deallocation function does not match the allocation function used for the same object.	Use of `free()` on an object allocated with `new`.
Erroneous	There is a bug is in the implementation of the operation.	Allocation with `malloc()` returns a non existing address.

data type or a data structure. Comments could be used to provide details, such as the object data type and identifier.

All possible causes for memory bugs are defined in Table II and Table IV. However, refer Fig. 2, Fig. 3, Fig. 4, and Fig. 5 for causes applicable to each class.

When describing a chain of bugs/weaknesses, the pointer and the object should be analyzed carefully, as they may be different for each improper state. The description should reflect the changes and provide details in the comments.

C. Consequences

A consequence is either an improper operand or a final error. As a consequence, an improper pointer or an improper object would become a cause for a next weakness. These consequence–cause transitions explain why these two kinds of consequences have the same possible values as the corresponding kinds of causes (see Table II and Table IV).

All possible memory errors are defined in Table V.

The only kind of MAD consequences is Improper Pointer, which means a MAD bug or weakness is always followed by another memory weakness, such as of MAL, MUS, or MDL. The only kind of MUS consequences is Memory Error, which means MUS always ends in a failure.

All possible consequences for memory bugs are defined in Table IV and Table V. However, refer Fig. 2, Fig. 3, Fig. 4, and Fig. 5 for consequences applicable to each class.

D. Attributes

An *attribute* provides additional useful information about the operation or its operands. All possible attributes for memory bugs are defined in Table VI.

All Memory Bugs classes have the following attributes: *Source Code*, *Execution Space*, and *Location*. They explain

TABLE III: Operands

Concept	Definition
Object	A memory region used to store data.
Pointer	A holder of the memory address of an object.

TABLE IV: Improper Operands

(a) Improper Pointer

Value	Definition
NULL Pointer	Points to the zero address, a specific invalid address.
Wild Pointer	Points to an arbitrary address, because it has not been initialized or an erroneous allocation routine is used.
Dangling Pointer	Still points to the address of its successfully deallocated object.
Over Bounds	Points over the bounds of its object.
Under Bounds	Points under the bounds of its object.
Untrusted Pointer	The pointer is modified to an improperly checked address.
Wrong Position	Points to a miscalculated position inside object bounds.
Hardcoded Address	The pointer points a wrong specific address.
Casted Pointer	The pointer does not match the type of the object, due to wrong type casting.
Forbidden Address	The pointer points to an OS protected or non-existing address.
Single Owner of Object	The only pointer of an already allocated object is used to allocate a new object.

(b) Improper Object

Value	Definition
Not Enough Allocated	The allocated memory is too little for the data it should store.
Wrong Size Used	The value used as size does not match the real size of the object.

TABLE V: Memory Errors

Value	Definition	Risk
Memory Overflow	More memory requested than available.	Stack/heap exhaustion.
Memory Leak	An object has no pointer pointing to it.	Resource exhaustion. Application crash. DoS.
Double Free	Attempt to deallocate a deallocated object or via an uninitialized pointer.	Arbitrary code execution.
Object Corruption	Object data is unintentionally altered.	Wrong/unexpected results.
Uninitialized Object	Object data is not filled in before use.	Controlled or left over data.
Not Cleared Object	Object data not overwritten before deallocation.	Information exposure (e.g. private keys).
NULL Pointer Dereference	Attempt to access an object for read or write through a NULL pointer.	Program crash. Arbitrary code execution (in some OSs).
Untrusted Pointer Dereference	Attempt to access an object via an altered pointer (not legitimate dereference of tainted pointers).	DoS. Arbitrary code execution.
Type Confusion	Pointer and object have different types.	Vtable corruption. Hijack.
Use After Free	Attempt to use a deallocated object.	Arbitrary code execution.
Buffer Overflow	Read or write above the object upper bound.	Arbitrary code execution. Information exposure.
Buffer Underflow	Read or write below the object lower bounds.	Arbitrary code execution. Information exposure.
Unitialized Pointer Derefereance	An attempt to access an object for read or write via an uninitialized pointer.	Control flow hijack.

where a bug is in three dimensions: where is the operation in the program, where its code is running, and where the object is stored in memory. See definitions of values in Table VIa.

All Memory Bugs classes have also the operation attribute *Mechanism*, but with different possible values.

For MAD and MUS *Mechanism* qualifies an operation as *Direct* or *Sequential*, depending on if an object element is accessed directly or after going through previous elements. See definitions of values in Table VIb.

For MAL and MDL *Mechanism* qualifies an operation as *Implicit* or *Explicit*. For MAL, *Implicit* means automatic compile-time allocation. Improper results from implicit allocation are not enough memory allocated or too much memory requested, overflowing the stack (e.g., via a recursion). For MDL, *Implicit* means automatic deallocation at the end of scope. Bugs in automatic memory allocation or deallocation are rare (e.g., the gcc compiler bug [6]). For MAL, *Explicit* means dynamic run time allocation (e.g. using `malloc()` or `new`). For MDL, *Explicit* means dynamic run time deallocation (e.g. using `free()` or `del`).

MAL and MDL have also the pointer attribute *Ownership*. It shows how many pointers point to an object: *None*, *Single*, and *Shared*. See definitions of values in Table VIc. For MAL, it shows how many pointers hold the allocated object. For MDL, if an object has no pointer pointing to it, it will be unreachable for deallocation in an environment without a garbage collector. Multiple pointers to the same object could lead to race conditions and dangling pointers.

MUS has also the pointer attribute *Span*. It shows how many bytes are being used: *Little*, *Moderate*, *Huge*, depending on if those are a few, more than a few and less than one KB, or more than one KB. See definitions of values in Table VId.

TABLE VI: Attributes

(a) MAD, MAL, MUS, MDL Attributes

Name	Value	Definition
Source Code	Codebase	The operation is in 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 executable (compiler, assembler, interpreter).
Execution Space	Userland	The bugged code runs in an environment with privilege levels, but in unprivileged mode (e.g., ring 3 in x86 architecture).
	Kernel	The bugged code runs in an environment with privilege levels with access privileged instructions (e.g., ring 0 in x86 architecture).
	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.
Location [1]	Stack	The object is a non-static local variable (defined in a function, a passed parameters, or a function return address).
	Heap	The object is a dynamically allocated data structure (e.g., via `malloc()` and `new`).

(b) MAD and MUS Attributes

Name	Value	Definition
Mechanism	Direct	The operation is performed over a particular object element.
	Sequential	The operation is performed after iterating over the object elements.

(c) MAL and MDL Attributes

Name	Value	Definition
Mechanism	Implicit	The operation is performed without a function call.
	Explicit	The operation is performed by a function/ method) call.
Ownership	None	The object has no owner.
	Single	The object has one owner.
	Shared	The object has more than one owner.

(d) MUS Attributes

Name	Value	Definition
Span	Little	A few bytes of memory are accessed.
	Moderate	Several bytes of memory are accessed, but less than 1 KB.
	Huge	More than 1 KB of memory is accessed.

E. Sites

MAD sites are any changes to a pointer via assignment (=) or repositioning via an index (`[]`) or pointer arithmetics (e.g., `p++` and `p--`).

[1]Other proper values should be used for different kinds of memory layout. For example, Uninitialized Data Segment, Data Segment, and Code Segment layout should be added for C language layout [7].

MAL sites are any allocation routine (e.g., `malloc()`) or operator (e.g. `new`), declaration of a variable with implicit allocation, OOP constructor, or extension routine (e.g., `realoc()`) or adding elements to a container object.

MUS sites are any dereference operators in the source code (`*`, `[]`, `->`, `.`).

MDL sites are any deallocation routine (e.g., `free()`) or operator (e.g. `del`), end of scope for implicit allocated variables, OOP destructor, or reduction routine (e.g., `realoc()`) or removing elements from a container object.

V. The BF Memory Classes as CWE Extension

BF Memory Bugs taxonomy can be used by bug reporting tools, as it is a structured extension over memory-related CWEs [2]. All Memory Error consequences from the BF classes (Table V) relate to one or more CWEs.

We have generated a digraph (Fig. 6) of all memory-related CWEs to show how they correspond to the possible BF Memory Error consequences (Table V). An edge starts at a parent CWE and ends at a child CWE. The outline of a CWE node indicates the CWE abstraction level: triple line is for variant, double line is for base, single line is for class, and thick red line is for pillar. Bug reporting tools would use base or variant CWEs, but they may also use higher abstraction level CWEs if there is not enough specific information about the bug or if there is no related base CWE.

The digraph helped us identify clusters of memory-related CWEs. All these CWEs can be tracked as children of the pillar CWE-664, with the only exception of CWE-476 (NULL Pointer Dereference). The largest cluster comprises CWE-118 and the children of CWE-119, which are weaknesses associated with reading and writing outside the boundaries of an object. The second cluster comprises the children of CWE-400 and CWE-665, which are mainly weaknesses related to memory allocation and object initialization. The children of CWE-404, which are weaknesses associated with improper memory cleanup and release, form the third cluster. The smallest cluster comprises CWE-704, CWE-588 and CWE-843, which are memory use or deallocation weaknesses due to the mismatch between pointer and object types.

The color of a CWE node (Fig. 6) indicates the BF memory class associated with that CWE. A BF class is associated with a CWE if the BF class has a Memory Error consequence covered by the CWE description. CWEs related to the BF MUS memory errors are presented in blue, CWEs related only to MAL are presented in pink, and CWEs related to both MAL and MDL are presented in green.

Most of the BF MUS Memory Error consequences (Fig. 4) relate to CWEs from the CWE-118 cluster. The Memory Error consequences from BF MAL and BF MDL (Fig. 3 and Fig. 5) relate to CWEs across clusters. Note that the BF MAD class (Fig. 2) has no Memory Error consequences, so it does not directly relate to any CWE.

The BF Memory Bugs model (Fig. 1) reflects the lifecycle of an object. While the pillar CWE-664 reflects the "lifetime of creation, use, and release" of a resource, it is quite broad. It is the parent of many CWEs that are not strictly memory-related. We use asterisks (*) to denote CWEs that are about any resource. CWE-704 is not a memory-related CWE, but is visualized on the digraph to show all the parent-child relationships.

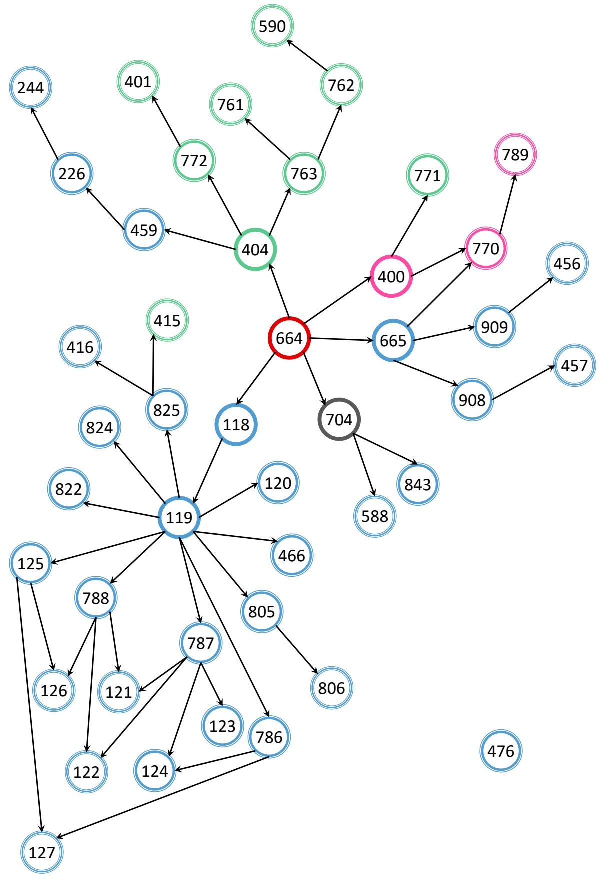


Fig. 6: A digraph of all memory related CWEs. Triple line – variant, double line – base, single line – class, and thick red line – pillar. BF class correspondence: pink is only for MAL, green is for both MDL and MAL, and blue is for MUS.

The identified clusters of memory CWEs do not strictly correspond to the phases of address formation, allocation, use, and deallocation. CWEs related to a phase appear in more than one cluster. In addition, CWE-118 and CWE-119 are strictly about memory but cover more than one phase.

Viewed as a structured extension, the BF Memory Bugs classes relate to CWEs through particular Memory Error consequences. For BF MAL: Memory Overflow – relates to CWEs: 400*, 770*, and 789; Memory Leak – to CWEs: 401, 404*, 771*, and 772*; Double Free – to CWE-415; Object Corruption – to CWEs: 404*, 590, 761, 762, and 763.

For BF MUS: Uninitialized Object – relates to CWEs: 456, 457, 665*, 908*, and 909*; Not Cleared Object – to CWEs: 226*, 244, and 459*; NULL Pointer Dereference – to CWE-476; Untrusted Pointer Dereference – to CWEs: 119 and 822; Type Confusion – to CWEs: 588 and 843*; Use After Free – to CWEs: 119, 416, and 825; Buffer Overflow – to CWEs: 118, 119, 120, 121, 122, 123, 125, 126, 466, 805, 806, 787, and 788; Buffer Underflow – to CWEs: 118, 119, 122, 123, 124, 125, 127, 466, 786, 787, 805, and 806; Unitialized Pointer Dereference – to CWEs: 119 and 824. There are no related CWEs to BF MUS Object Corruption.

For BF MDL: Memory Leak – relates to CWEs: 401, 404*, and 771*; Double free – to CWE-415; Object Corruption – to CWEs: 404*, 761, 762, and 763.

VI. Showcases and Discussion

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

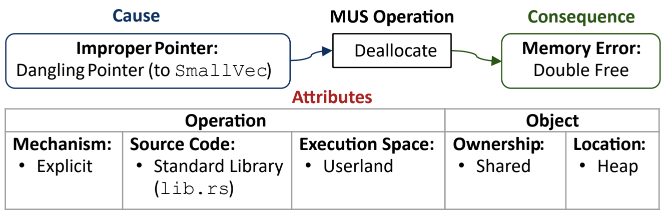
A. *CVE-2018-20991 – Rust SmallVec Iterator Panic*

This vulnerability is listed in CVE-2018-20991 and discussed in [8]. The source code could be found at [9]. In Rust, a panic is an unrecoverable error that terminates the thread, possibly unwinding its stack (calling destructors as if every function instantly returned) [10].

a) Brief Description: Rust is a multi-paradigm programming language focused on safe concurrency. It has a similar syntax to C++ and offers features to deal with dynamic memory allocation, such as smart pointers [11]. In general, a Rust programmer does not need to keep track of memory allocation and deallocation, as the language is designed to be memory safe this way.

b) Analysis: The versions before Rust 0.6.3 have a bug in the `lib.rs` file. The `insert_many()` method in the `SmallVec` class has two parameters: an iterable `I` and an `index`. The method inserts all elements in the iterable `I` at position `index`, shifting all the following elements backwards. In the `SmallVec` class, if an iterator passed to `SmallVec::insert_many()` panics in `Iterator::next`, the destructor is called while the vector is in an inconsistent state, possibly causing double free (deallocation via references to same object). Fig. 7 presents the BF taxonomy for this vulnerability.

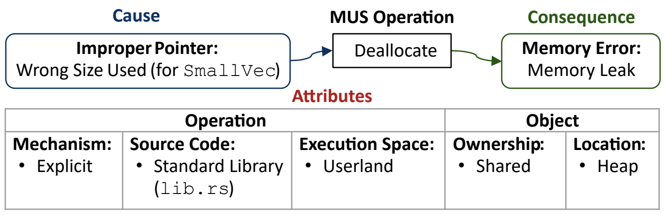
c) The Fix: To fix the bug, the Rust community opted to set the `SmallVec` length to `index`, call `insert_many()`, and then update the length. With this fix, if an iterator panics, a memory leak occurs [12]. The developers downgraded the bug to avoid double free as a consequence, which could lead to arbitrary code execution. Now they have a memory leak. Fig. 8 presents the BF taxonomy for the new bug.



Cause
Improper Pointer: Dangling Pointer (to SmallVec)
MUS Operation
Deallocate
Consequence
Memory Error: Double Free
Attributes

Operation			Object	
Mechanism:	Source Code:	Execution Space:	Ownership:	Location:
• Explicit	• Standard Library (`lib.rs`)	• Userland	• Shared	• Heap

Fig. 7: BF for CVE-2018-20991 – Rust Iterator Panic



Cause
Improper Pointer: Wrong Size Used (for SmallVec)
MUS Operation
Deallocate
Consequence
Memory Error: Memory Leak
Attributes

Operation			Object	
Mechanism:	Source Code:	Execution Space:	Ownership:	Location:
• Explicit	• Standard Library (`lib.rs`)	• Userland	• Shared	• Heap

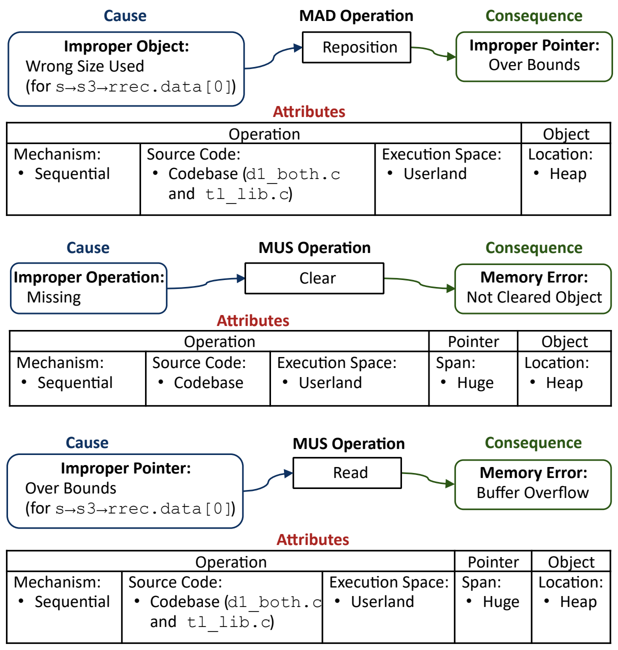
Fig. 8: BF for the Bug in the Fix of CVE-2018-20991

B. *CVE-2014-0160 – Heartbleed Buffer Overflow*

This vulnerability is listed in CVE-2014-0160 and discussed in [13]. The source code could be found at [14].

a) Brief Description: Heartbleed is a vulnerability due to a bug in the OpenSSL – a crypto library for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. Using the heartbeat extension tests in TLS and Datagram Transport Layer Security (DTLS) 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. The server responds with the same string. However, due to the bug, a malicious user could set the `payload` as big as `65535` and the server would read out of bounds. This could expose confidential information that was not cleared before release.

b) Analysis: The TLS and DTLS implementations in OpenSSL 1.0.1 before 1.0.1g have a bug in the `d1_both.c` and `t1_lib.c` files. In the Heartbleed attack, the software stores the user data in an array `s->s3->rrec.data[0]`. The size of that array is much less than the huge `65535` bytes payload. The software does not check the size of the data (`s->s3->rrec.length`) towards the value of the payload. It assumes these numbers are equal and using `memcpy()` reads `payload` consecutive bytes from the array, beginning at its first byte, then sends them to the malicious user. Fig. 9 presents the BF taxonomy for this vulnerability.



Cause
Improper Object: Wrong Size Used (for s→s3→rrec.data[0])
MAD Operation
Reposition
Consequence
Improper Pointer: Over Bounds
Attributes

Operation			Object
Mechanism:	Source Code:	Execution Space:	Location:
• Sequential	• Codebase (`d1_both.c` and `t1_lib.c`)	• Userland	• Heap

Cause
Improper Operation: Missing
MUS Operation
Clear
Consequence
Memory Error: Not Cleared Object
Attributes

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

Cause
Improper Pointer: Over Bounds (for s→s3→rrec.data[0])
MUS Operation
Read
Consequence
Memory Error: Buffer Overflow
Attributes

Operation			Pointer	Object
Mechanism:	Source Code:	Execution Space:	Span:	Location:
• Sequential	• Codebase (`d1_both.c` and `t1_lib.c`)	• Userland	• Huge	• Heap

Fig. 9: BF for CVE-2014-0160 – Heartbleed Buffer Overflow

c) The Fix: To fix the bug the openSSL team added a bound check for the array size [15]. We should note that in Fig. 9 the Wrong Size Used cause is a consequence from a missing Verify operation of a preceding Data Verification Bug (DVR) [1], which is beyond the scope of this paper.

C. Discussion

The BF taxonomy of a vulnerability can help identify exploit mitigation techniques for a particular weakness types. For that we should connect the BF taxonomy to an appropriate attack model.

For memory bugs, we can use the classic memory corruption attack model of Szekeres et al. [16] that systematizes the memory protection techniques. The model has six steps towards the ultimate goal of an attacker. Its very first level is on memory safety, where an attacker can start an exploitation with an invalid pointer dereference. This kind of invalid pointer corresponds to the improper pointer states that define some of the causes for the BF Memory Bugs classes (Table IVa).

Using the BF description of a vulnerability and following the attack model we can identify effective mitigations against possible attacks. Let's take, for example, a Buffer Overflow that is caused by Read Over Bounds. Following the Szekeres et al. model, such a bug would allow an attacker to access program data, leading to information leakage.

To make use of the collected data, the attacker should be able to interpret it. Probabilistic methods such as data space randomization (DSR) could mitigate the attack, while an address space location randomization (ASLR) will not do it [17]. The values of the Location and Execution Space attributes of the object help identify where in the memory layout the mitigation technique should be put in place.

The Szekeres et al. model, however does not cover bugs related to some BF memory operations, such as allocation, reallocation, and initialization. It does not cover any memory addressing bugs (MAD) and it is not concerned describing how a pointer becomes invalid. A key point here is that Szekeres et al. look at memory corruption bugs from attacks perspective, while we focus on systematizing information that is sufficient to fix a bug.

VII. Conclusion

In this paper, we introduce four new BF classes: Memory Addressing Bugs (MAD), Memory Allocation Bugs (MAL), Memory Use Bugs (MUS), and Memory Deallocation Bugs (MDL). We present their operations, along with the possible causes, consequences, attributes, and sites.

We analyze particular vulnerabilities related to these classes and provide precise BF descriptions. The BF structured taxonomies of memory corruption vulnerabilities show the initial error (the bug) providing a quite concise and still far more clear description than the unstructured explanations in current repositories, advisories, and publications.

Linking the BF Memory Bugs model and taxonomy to an attack model (e.g. Szekeres et al. model) would provide the means of covering the memory corruption vulnerabilities landscape. For example, the first layer of the Szekeres model could connect with the BF causes defined in Section IV-B. As part of that, the notion of invalid pointer should not be restricted to dangling pointers and out of bounds pointers; refinement to the causes in Table IVa should be considered.

The BF Memory Bugs taxonomy can be used by bug reporting tools, as it can be viewed as a structured extension over the memory related CWEs [2]. Furthermore, the BF descriptions of particular vulnerabilities can be used to identify exploit mitigation techniques.

References

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

[2] MITRE, *Common weakness enumeration (CWE)*, Accessed: 2020-02-29, 2020. [Online]. Available: https://cwe.mitre.org.

[3] ——, *Common vulnerabilities and exposures (CVE)*, Accessed: 2020-02-29, 2020. [Online]. Available: https://cve.mitre.org/.

[4] NVD, *National vulnerability database (NVD)*, Accessed: 2020-01-10, 2020. [Online]. Available: https://nvd.nist.gov.

[5] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. of ISSTA*, ACM, 2012, pp. 133–143.

[6] F. Heckenbach, Accessed: 2020-02-29, 2015. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66139.

[7] V. Vikas, *Memory layout of C for beginners*, Accessed: 2021-02-18, 2019. [Online]. Available: https://medium.com/@vikasv210/memory-layout-in-c-fe4dffdaeed6.

[8] *RUSTSEC-2018-0003*, Accessed: 2020-02-29, 2018. [Online]. Available: https://rustsec.org/advisories/RUSTSEC-2018-0003.html.

[9] *rust-smallvec - Panic-safety fixes #103*, Accessed: 2020-02-29, 2014. [Online]. Available: https://github.com/servo/rust-smallvec/pull/103/files.

[10] The Rust Team, *The rustonomicon*, Accessed: 2020-04-06, 2020. [Online]. Available: https://doc.rust-lang.org/nomicon/unwinding.html.

[11] G. Hoare, *Rust*, Accessed: 2020-02-29, 2020. [Online]. Available: https://www.rust-lang.org/.

[12] *SmallVec::insert_many is unsound #96*, Accessed: 2020-02-29, 2018. [Online]. Available: https://github.com/servo/rust-smallvec/issues/96.

[13] *The heartbleed bug*, Accessed: 2020-02-29, 2014. [Online]. Available: https://heartbleed.com/.

[14] *OpenSSL*, Accessed: 2020-02-29, 2014. [Online]. Available: https://git.openssl.org/gitweb/?p=openssl.git;a=blob;f=ssl/d1_both.c;h=0a84f957118afa9804451add380eca4719a9765e;hb=4817504d069b4c5082161b02a22116ad75f822b1.

[15] *OpenSSL*, Accessed: 2020-02-29, 2014. [Online]. Available: https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3.

[16] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. of 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[17] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," in *Proc. of 12th European Conf. on Computer Systems*, ACM, 2017.