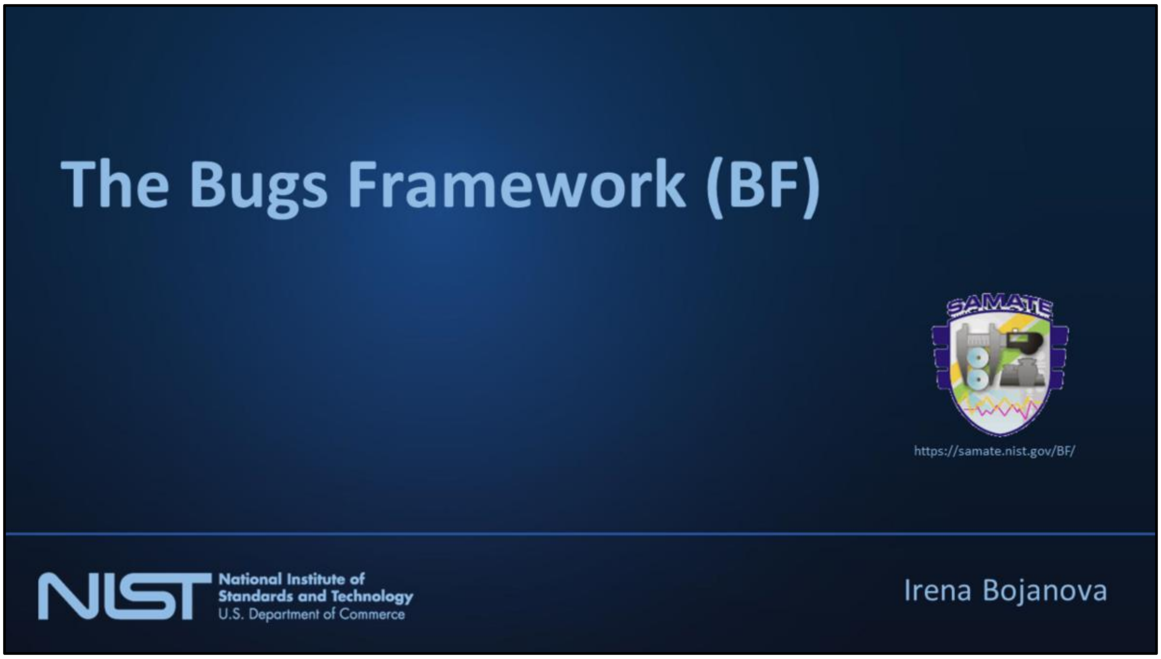


Welcome to the NIST Bugs Framework presentation!
I am Irena Bojanova – a Computer Scientist at NIST and the Primary Investigator and Lead of this project.

We are classifying software bugs and weaknesses to allow precise descriptions of vulnerabilities that exploit them.

Agenda

NIST

- Terminology:
  - Bug, Weakness
  - Vulnerability
  - Failure
- Existing Repositories:
  - CWE
  - CVE
  - NVD
- The Bugs Framework (BF)
  - Goals
  - Features
- Example – Heartbleed
- Potential Impacts

In this presentation, I will define key notions for BF,
discuss the commonly used repositories
of software weaknesses and vulnerabilities,
and present BF's goals, features, and potential impacts.

I will showcase BF by describing the Heartbleed vulnerability.

We need strict definitions of the software terms:
bug, weakness, and vulnerability.

Bug, Weakness, Vulnerability, Failure

NIST

- Software Bug:
  - A coding error
  - Needs to be fixed
- Software Weakness – **difficult to define**:
  - Caused by a bug or ill-formed data
  - Weakness Type – a meaningful notion!
- Software Vulnerability:
  - An instance of a weakness type that leads to a security failure
  - May have several underlying weaknesses

We define a software bug
as a coding error that needs to be fixed.

<click 1>

- We know that 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.

<click 2> 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.

Existing Repositories

There are several commonly used repositories of software weaknesses and vulnerabilities.

Commonly Used Repositories

NIST

- Weaknesses:
  CWE – Common Weakness Enumeration

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

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

We focus on the following:

- CWE is a community-developed list of software and hardware weaknesses types.
- CVE is a catalog of publicly disclosed cybersecurity vulnerabilities.
- NVD is the US government repository that links all CVEs to CWEs.

Repository Problems

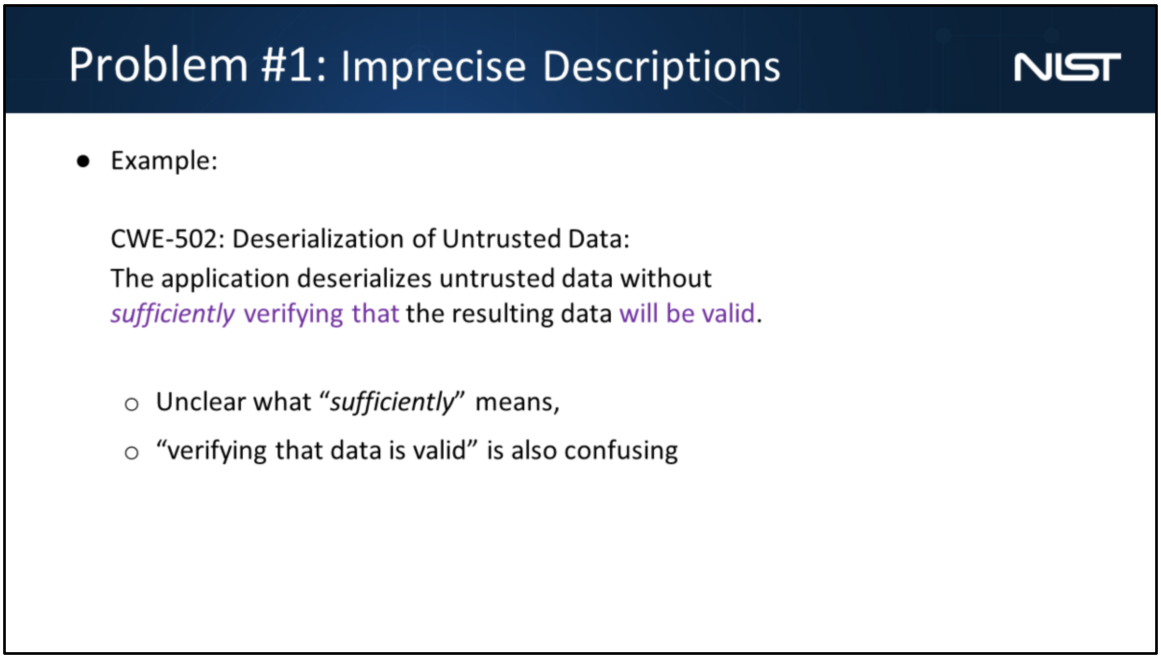
NIST

1. Imprecise Descriptions – CWE & CVE
2. Unclear Causality – CWE & CVE
3. Gaps in Coverage – CWE
4. Overlaps in Coverage – CWE

CWE and CVE are widely used,
but they have some problems.

Many CWEs and CVEs have
imprecise descriptions and unclear causality.

CWE also has gaps and overlaps in coverage.



This is an example of an imprecise CWE definition.

It states: "The application deserializes untrusted data without
sufficiently verifying that the resulting data will be valid."

It's not clear here what "sufficiently" means;
and "verifying that data is valid" is also confusing;
It should say "... without validating and verifying it".

Problem #2: Unclear Causality

- Example:

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

→ the NVD label is CWE-190

While the CWEs chain is:
CWE-20 → CWE-190 → CWE-119

Unclear causality in CVEs leads to wrong CWE assignments.

For example, in this CVE,
luck of input validation leads to
integer overflow and then to buffer overflow.

NVD labels it with CWE-190 – Integer Overflow or Wraparound,
while the cause is CWE-20 – Improper Input Validation.

The full chain is: CWE 20 – CWE 190 – CWE 119
the last one being – Improper Restriction of Operations
within the Bounds of a Memory Buffer.

Problems #3, #4: Gaps/Overlaps in Coverage

NIST

- Example:

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

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

Gaps and overlaps in CWEs lead to confusion.

As an example, if we arrange buffer overflow CWEs
by read or write,
over or under the bounds,
on the stack or heap,

<click 1> the gaps and overlaps can be easily spotted.

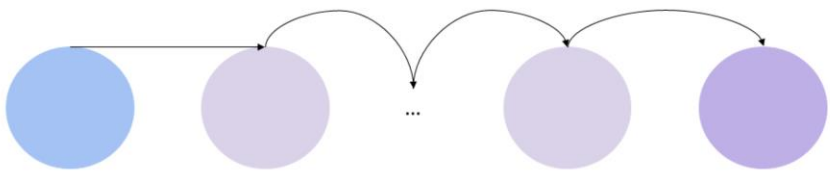
The Bugs Framework (BF)

The Bugs Framework aims to address
all these CWE and CVE problems.

It should have the expressiveness power
to clearly describe
any software bug or weakness,
underlying any vulnerability.

BF Goals

1. Solve the problems of imprecise descriptions and unclear causality

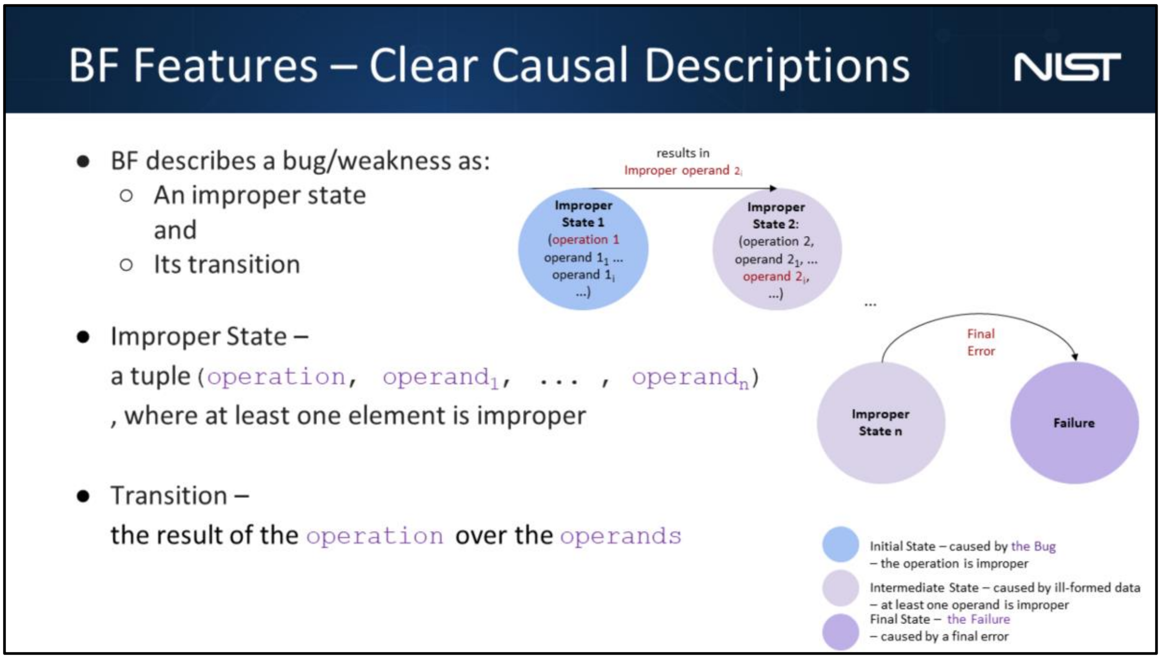


2. Solve the problems of gaps and overlaps in coverage

To solve the problems of
imprecise descriptions and unclear causality,
BF should be a structured classification.

<click 1> The BF description of a vulnerability
should provide causal relationships –
forming a chain of underlying weaknesses,
leading to a failure.

<click 2> To avoid gaps and overlaps,
BF should be a complete, orthogonal classification.



BF describes a bug or a weakness as:
<click 1> an improper state and its transition.
<click 2> The transition is to another weakness or to a failure.

<click 3> An improper state is defined by
a tuple – operation and its operands,
where at least one element is improper.

The initial state – depicted in blue –
is always caused by a bug –
a coding error within the operation,
which if fixed will resolve the vulnerability.

An intermediate state – in light purple –
is caused by ill-formed data –
it has at least one improper operand.

The final state, the failure – in dark purple –
is caused by the final error –
undefined or exploitable system behavior.

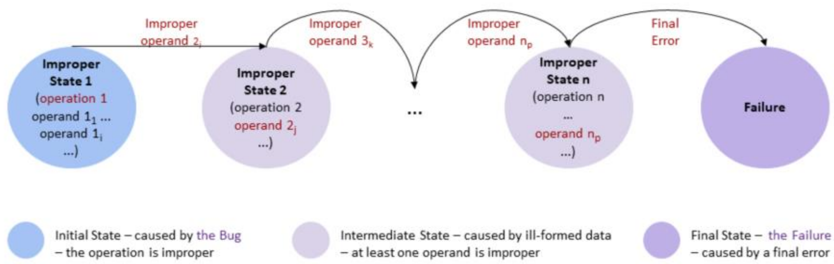
A transition is the result of the operation over the operands.

For example, here,
<click 4> Improper Operation 1 from Improper State 1 results in Improper Operand 2_i leading to Improper State 2.
<click 5> The last operation results in a Final Error, leading to a failure.

BF Features – Chaining Weaknesses

NIST

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

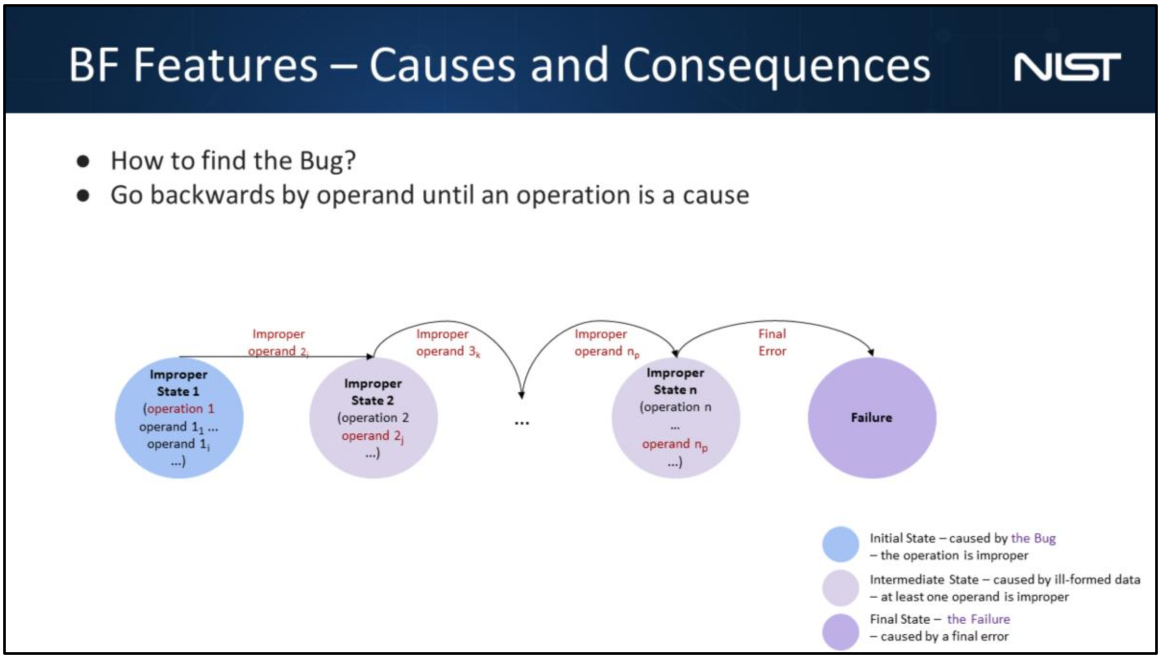


BF describes a vulnerability as
a chain of improper states and their transitions.
Each improper state is an instance of a BF class.

<click 1> The transition from the initial state is
by improper operation over proper operands.

<click 2> The transitions from intermediate states are
by proper operations with at least one improper operand.

<click 3> In rare cases an intermediate state may also have a bug,
which if fixed will also resolve the vulnerability.

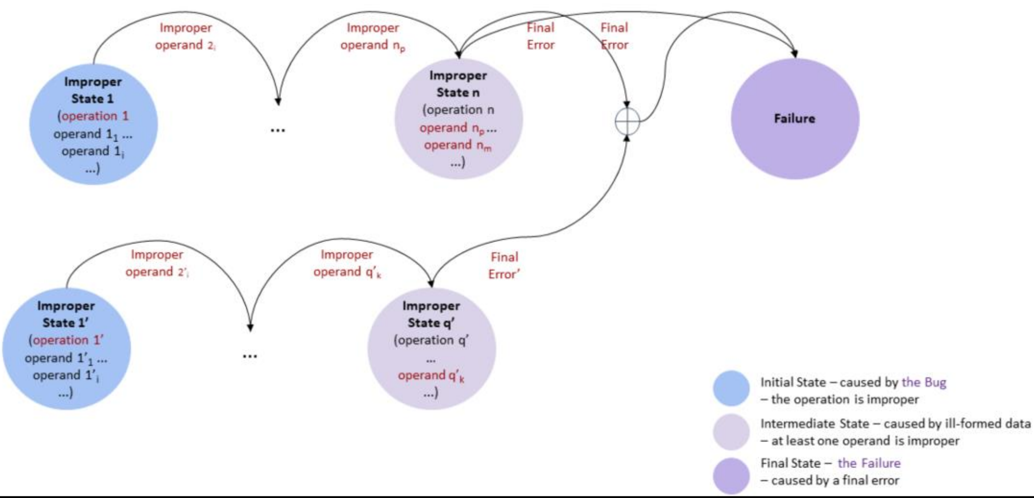


<click 1> The improper operation or operand
is the cause for that weakness.

<click 2> The improper result from
an operation over its operands
is the consequence from that weakness,
<click 3> and it becomes a cause
for next weakness or a failure.

Knowing the failure and
all the transitions at execution,
we should be able to find the bug,
<click 4> Simply go backwards by operand
until an operation is a cause –
fixing the bug within that operation
will resolve the vulnerability.

BF Features – Converging Vulnerabilities



<click 1> In some cases, several vulnerabilities have to be present for an exploit to be harmful.

<click 2> 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.

BF Features – Classification

NIST

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

BF'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.

Operations or operands
improperness define the causes.

A consequence is the result of
the operation over the operands.
It becomes the cause for

a next weakness or a failure.

The attributes describe
the operations and the operands.
They help us understand
the severity of the bug.

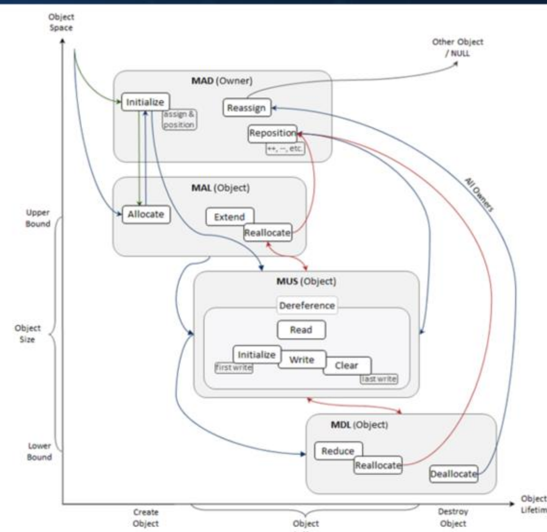
BF – Bugs Models

NIST

- Example:

  The BF Memory Bugs Model:

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

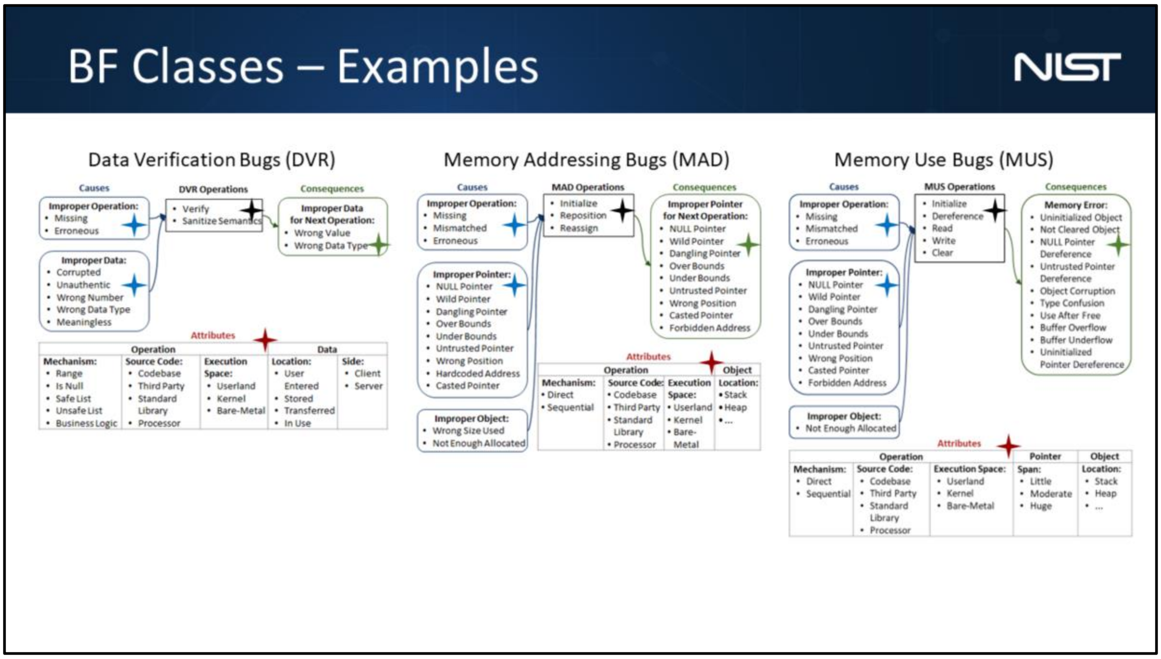


We create bugs models
to help us identify the BF classes.

They show the phases,
where particular types of bugs could occur,
and the possible flow of operations.

For example,
the memory bugs model shows
the identifies phases and operations
for memory addressing, allocation,
use, and deallocation bugs.

It assures the corresponding BF classes
MAD, MAL, MUS, and MDL
do not overlap in operations.



Data Verification Bugs (DVR),
Memory Addressing Bugs (MAD)
and the Memory Use Bugs (MUS)
are three of our fully developed BF classes.

<click 1> Each has a set of operations –
where such bugs could happen;
<click 2> a set of causes –
the possible improper operations and operands,
<click 3> a set of consequences –
improper operands for next weakness
and the possible failures
<click 4> and a set of attributes with values –
for the operations and the operands.

BF – Defined

NIST

- BF is a …

  - ➢ Structured
  - ➢ Complete
  - ➢ Orthogonal
  - ➢ Language independent

  classification of software bugs and weaknesses

We define BF as
a structured, complete, orthogonal
classification of software bugs and weaknesses,
which is also "language independent".

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 assures precise causal description.

Complete means:
BF has the expressiveness power
to describe any software bug or weakness
→ this solves the gaps problem

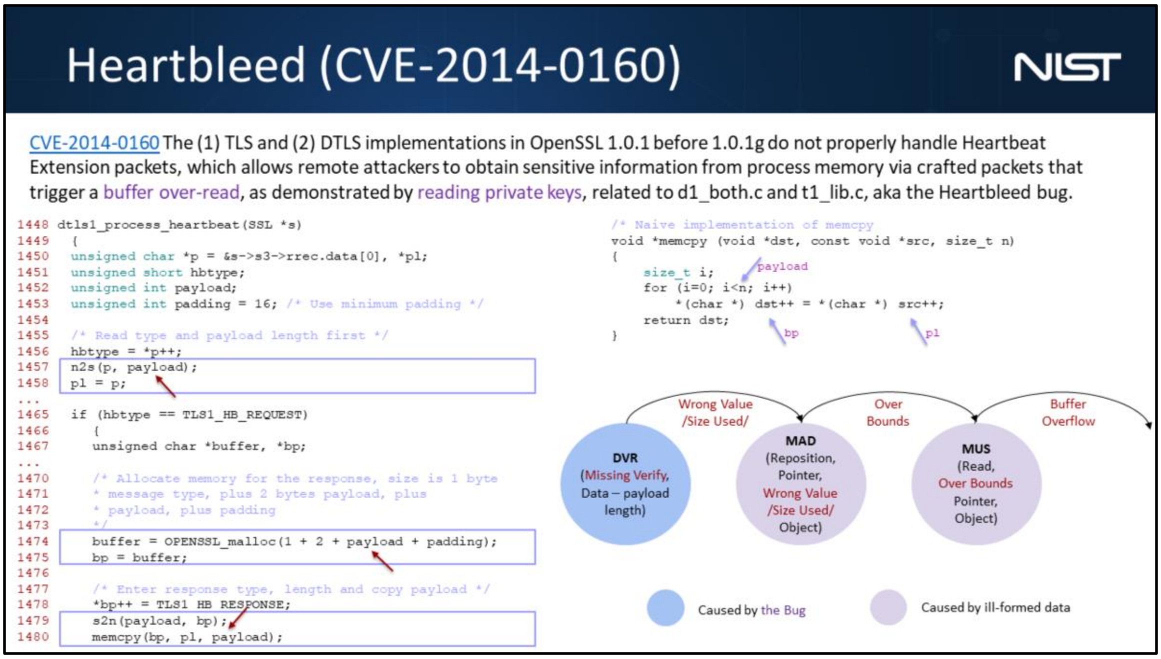
Orthogonal means:
the sets of operations
of any two BF classes do not overlap
→ this solves the overlaps problem.

BF is also applicable for source code

in any programming language.

BF Example – Description of Heartbleed

Let’s see now how BF is used
to describe real world vulnerabilities.



Heartbleed was a vulnerability in the OpenSSL cryptographic software library. The bug was in the TLS implementation of the heartbeat extension. It was disclosed in April 2014 with the following CVE: "The TLS and DTLS implementations in OpenSSL did not properly handle Heartbeat Extension packets, which allowed remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys."

<click 1> Let's examine the code. The problem is in the data verification phase, where the semantics of the input should be checked and sanitized.

<click 2> payload is a unsigned integer and can be a huge number. It is input data, that holds the payload length, but it's not checked towards a upper limit.

It's value is not verified!

<click 3> This improper state is an instance of the BF DVR class. The operation verify is missing.

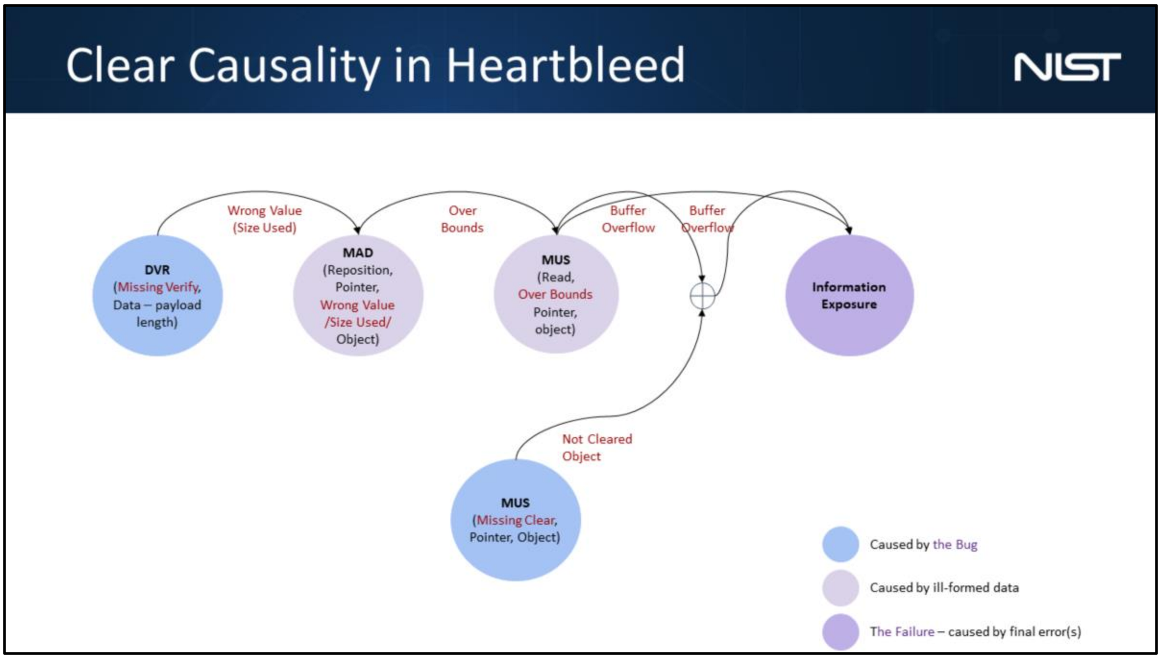
<click 4> memcopy reads payload number of bytes from the object pointed by "pl" and copies them to the object pointed by "bp". "pb" and "pl" are passed by reference via "dst" and "src";

<click 5> and the huge payload length is passed via the argument "n". First, one byte is read from "pl" and copied to "pb"; then until the huge payload length is reached, both pointers move one byte up and the newly pointed by "pl" byte is read and copied.

<click 6> While "bp" is allocated large enough, "pl" points to an array with reasonable size. As the content of this array is read and copied to "bp", so is also huge amount of data from over its bounds. There are two improper states here: when "pl" gets repositioned over it's upper bound and when data is read from there.

<click 7> The former is an instance of the BF MAD class. There is no bug in the repositioning itself,

however wrong value is used as size for the “pl” object.
<click 8> The latter is an instance of the BF MUS class. Again, there is no bug in the read operation itself, but because “pl” points over bounds, the software reads data
that should not be read, aka buffer overflow.



This chain of BF states shows there is buffer overflow, however, it does not show how an exploit could reach sensitive information, such as private keys.

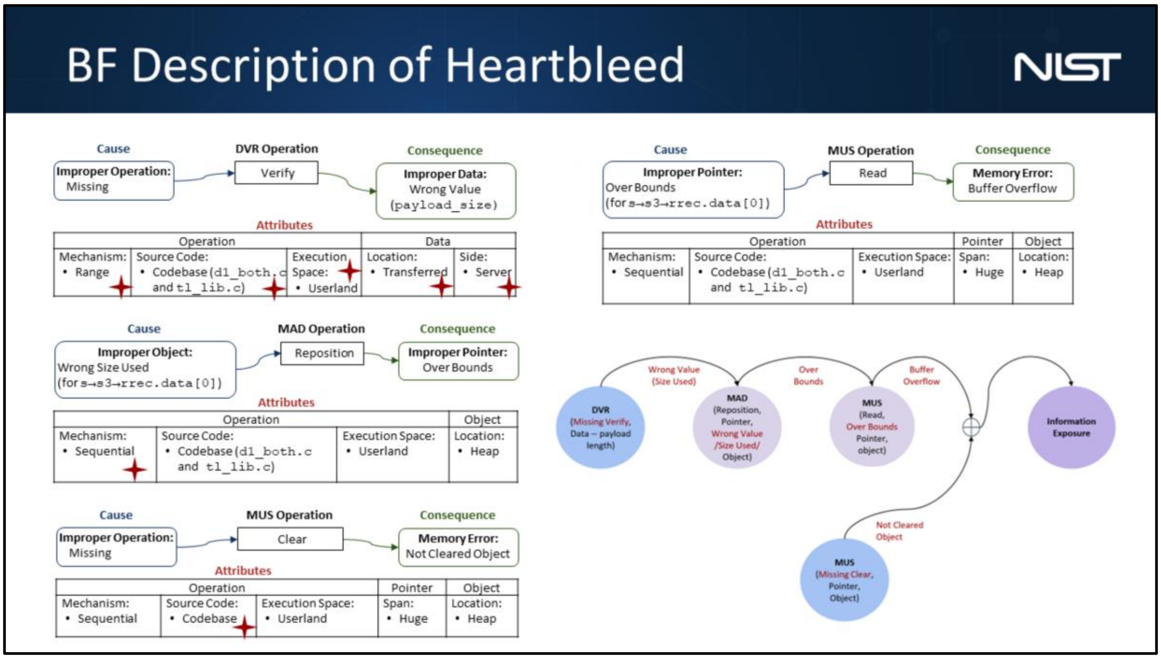
The missing size verification bug is not enough to get access to private data.

There must have been another coding error due to which, unaware of the risks, an unused object with sensitive data is left in memory.

<click 1> To describe the bug in this parallel vulnerability, we use again the BF MUS class, but this time the improper operation is missing clear.

Combining the final errors from both chains, the bugged software can now reach and expose sensitive information.

The BF description of Heartbleed is: Missing data verification leads to use of wrong size for an object, allowing a pointer reposition over bounds, which converging with missing clear allows reads of sensitive information and its exposure.

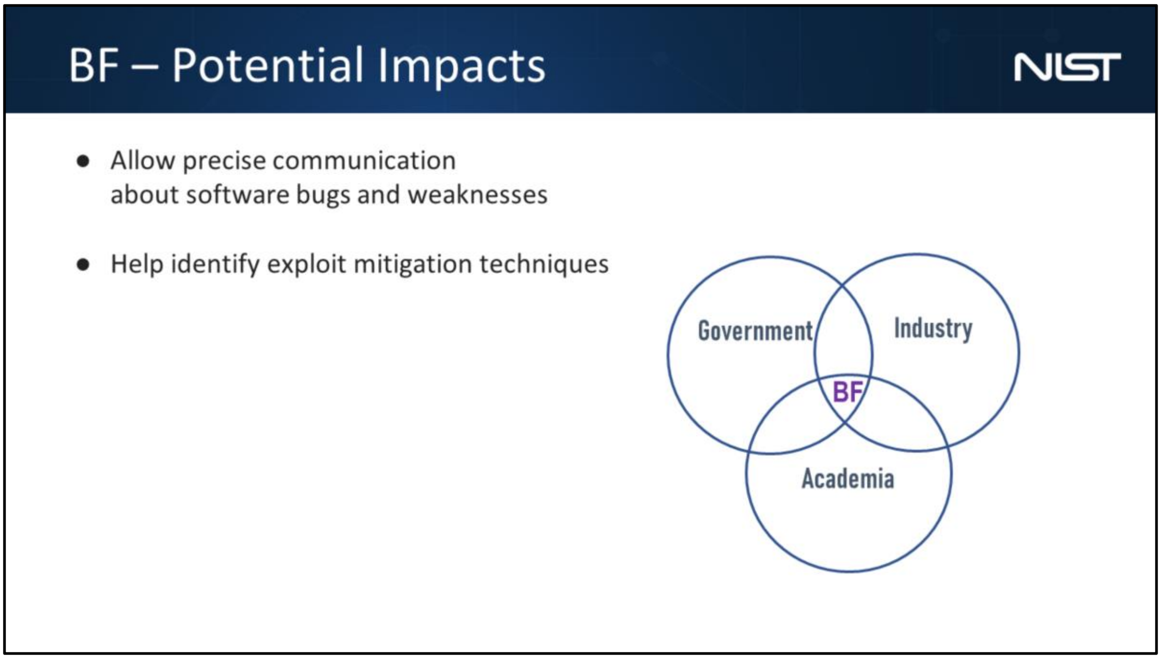


Using the BF taxonomy of the involved weaknesses, first is the data verification bug – DVR. Missing verification leads to wrong value.

The attributes show how and where this went wrong.
<click 1> Mechanism points the missing verification should have been – check against a range.
<click 2> Source code shows where the buggy code is in software.
<click 3> Execution space is about the privilege level.
<click 4> Location and side show where the data is.
<click 5> Next is the MAD weakness: Wrong size used at reposition leads to pointer over bounds.
<click 6> The mechanism attribute here shows how the reposition is done.
<click 7> Last in this vulnerability is the MUS weakness, which results in buffer overflow.
<click 8> Coming from another chain is again a MUS weakness: Missing clear leads to a not cleared object. The attributes are the same as for MAD.
<click 9> However, this is a different vulnerability and the source code in different software.
<click 10> The final errors buffer overflow and not cleared object, combined, cause Information Exposure.
<<END HeartBleed>>

BF – Potential Impact

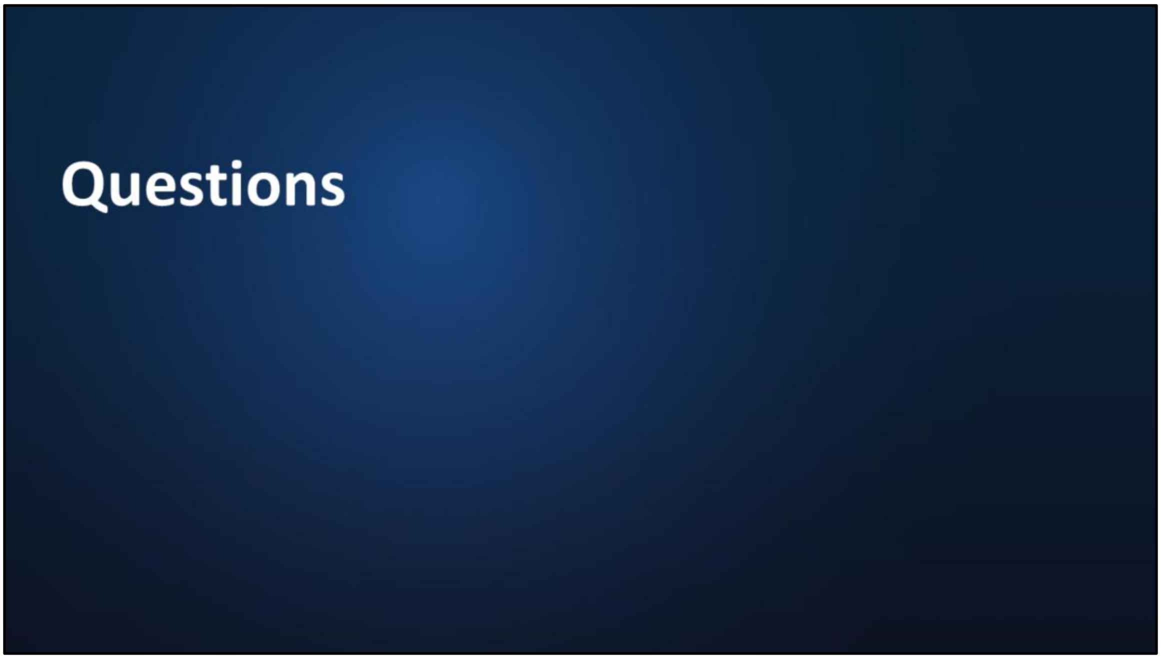
Why BF matters?



BF will allow precise communication
about software bugs and weaknesses
and will help identify
exploit mitigation techniques.

- Government could:
  - Improve the descriptions in public vulnerability repositories
  - Create policies and guidelines for software testing
- Software companies could:
  - Improve the testing tools and their bug reports
  - Implement automatic bugs finding and fixing.
- Professors could:
  - Teach better about bugs and weaknesses
  - Conduct research on formalizing software bugs.

<<END>>



Please do not hesitate to contact us
with questions and ideas for collaboration.

Questions

NIST

Irena Bojanova: irena.bojanova@nist.gov

SAMATE

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