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

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

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

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

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

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

Keywords—software weaknesses; bug taxonomy; attacks.

I. Introduction

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

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

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

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

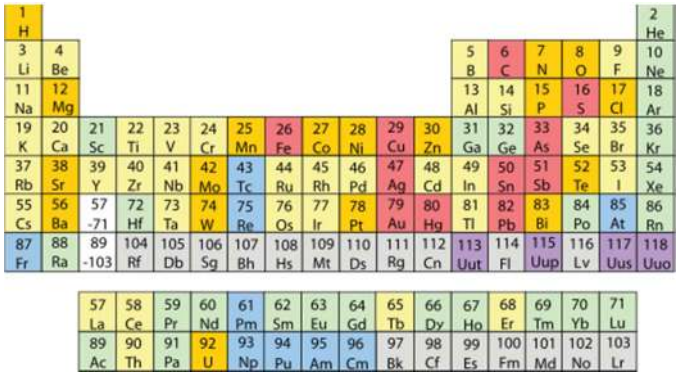

Fig. 1. Periodic Table of Elements: antiquity, Levoisier 1789, Mendeleev 1869, Deming 1923, Seaborg 1945, up to 2000, to 2012[1].

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 does it imply that they are necessarily the best available for the purpose.

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

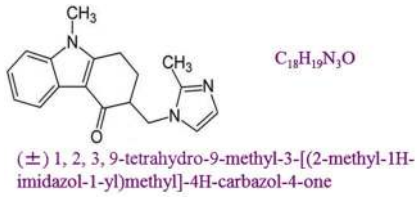

Fig. 2. Three ways to describe Zofran ODT.

classified using loops, whorls and arches and retrieved based on minutia. Chemists have a detailed system beyond the periodic table to describe chemicals. For instance, they have several different systems of rendering molecules, which are three dimensional, to emphasize aspects that are more important in different contexts, see Fig. 2.

Finally, all integers[2] have unique prime factors. Analogously, we seek to factor software weaknesses into their constituent components, thereby gaining the understanding to organize these components in their most naturally-occurring categories and structure. We aim for the most accurate, precise and intuitive way to describe software bugs.

To paraphrase William Thomson, Baron Kelvin, "when you can measure what you're speaking about, and express it [in precise terms], you know something about it; but when you cannot, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced it to the stage of *science*." [3]

In this paper, we first discuss existing software weaknesses enumerations, patterns and templates. Then we present our Bugs Framework (BF) with its four main areas: causes, attributes, consequences and sites. To make sure that BF applies to all classes of bugs, we began with three quite disparate classes: buffer overflows, injections and control of frequency of interactions. Buffer overflow occurs primarily in C and is low-level; injection relates strongly to the language in which the command string is interpreted, and control of frequency interactions requires reference to a user-level policy to set limits. For each class, we provide a definition and the BF taxonomy, which includes the sites in code where they may be found. We also provide examples of applying the taxonomy to describe particular vulnerabilities and list corresponding classes from other weaknesses collections. The final section summarizes our work and discusses the benefits from our BF as well as our future plans. Our goal is for the BF to become the software developer's and tester's "Best Friend."

II. Existing Enumerations, Patterns and Templates

The Common Weakness Enumeration (CWE) [1] is an "encyclopedia" of over 600 types of software weaknesses. Some of the classes are buffer overflow, directory traversal, OS injection, race condition, cross-site scripting, hard-coded password and insecure random numbers. CWE is a widely-used compilation, which has gone through many iterations. Many tools and projects are based on it. Each CWE has a variety of information, such as description summary, extended description, white box definition, consequences, examples, background details and other notes, recorded occurrences (Common Vulnerabilities and Exposures or CVE [4]), mitigations, relations to other CWEs, and references.

CWEs are a rich source of material for software developers and superior to anything that existed before. However, for very formal, exacting work, CWE definitions are often inaccurate, imprecise or ambiguous, and the various definitions within one CWE can be inconsistent. Each CWE bundles many stages, such as likely attacks, resources affected and consequences. The coverage is uneven, with some combinations of attributes well represented and others not appearing at all. An extreme instance is path traversal. There are a dozen CWEs for path traversal, each one having a specific combination of relative or absolute paths, forward or backward slashes – singly or repeated, between one and three directory steps, and two or more dots, which indicate the parent directory.

Another example is buffer overflows. CWE-121 [5] is write outside of a buffer on the stack, CWE-122 is write outside of a buffer in the heap, CWE-127 is read before the beginning of a buffer and CWE 126 is read after the end of a buffer. But there are no CWEs specifically for read outside a buffer on the stack vs. in the heap. The description summary of CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer is "The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer." Note that "read from or write to a memory location" is not explicitly tied to the buffer! Most humans would, of course, assume that it means the software can access through a buffer a memory location that is not allocated to that buffer.

Software Fault Patterns (SFP) [6] are a clustering of CWEs into related weakness categories. Each cluster is factored into formally defined attributes, with sites ("footholds"), conditions, properties, sources, sinks, etc. This work overcomes the problem of combinations of attributes in CWE. For instance, Table 1 shows how SFP factored attributes are more clear than the irregular coverage of CWEs.

SFP is an excellent advance, but does not tie fault clusters to causes or chains of fault patterns nor to consequences of a particular vulnerability. In addition, since they were derived from CWEs, more work is needed for embedded or mobile concerns, such as, battery drain, physical sensors (e.g. Global Positioning System (GPS) location, gyroscope, microphone, camera) and wireless communications.

Another source of organization of weaknesses is Semantic Templates (ST). "A semantic template is a human and machine understandable representation of the following: 1) software faults that lead to a weakness; 2) resources that a weakness affects; 3) weakness attributes; and 4) consequences/ failures resulting from the weakness." [7] Semantic Templates factor out chains of causes, resources and consequences that are present in CWEs. For instance, Fig. 3 shows phrases in the description summary, extended description and common consequences of CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), labeled according to the phases called out by Semantic Templates.

Details on the relevant body of knowledge that consolidates CWE, including the SFP and the ST efforts is presented in [8].

[2] Greater than one.

TABLE I. SFP FACTORED ATTRIBUTES OF BUFFER OVERFLOW CWES

CWE \ Attribute	Location		Access kind		Boundary exceeded	
	heap	*stack*	*read*	*write*	*lower*	*upper*
119: Improper Restriction of Operations within Bounds of Buffer	√	√	√	√	√	√
120: Buffer Copy without Checking Size of Input	√	√		√	√	√
121: Stack Overflow		√		√	√	√
122: Heap Overflow	√			√	√	√
123: Write-what-where condition	√	√		√	√	√
124: Buffer Underwrite	√	√	√	√	√	
125: Out-of-bounds read	√	√	√		√	√
126: Buffer Overread	√	√	√			√
127: Buffer Underread	√	√	√		√	

Landwehr et. al. created a taxonomy of security flaws in programs [2]. The taxonomy has three aspects: genesis, that is, how it originated, time of introduction and location. Each aspect is further divided into subcategories. The main focus of the taxonomy seems to be how flaws originated and is aimed at a higher, system level. It does not include details enabling automated detection in code, proving the efficacy of mitigation techniques or deriving possible consequences.

III. THE BUGS FRAMEWORK (BF)

Just as integers can be factored into prime numbers or molecules can be decomposed into constituent atoms, we break down information in CWEs, SFPs, and other compendia and collections into basic, orthogonal components.

We organize them into meaningful structures and identify rules of composition. We use this compilation in several ways in order to validate it and demonstrate its utility. We elucidate known vulnerabilities, accurately and precisely defining the classes of bugs reported by assurance tools and document in exactly what situation various software assurance techniques are efficacious. We believe this compilation may also guide development of techniques to cover gaps.

The BF comprises four main areas: causes, attributes, consequences and sites of bugs. The causes and consequences are well represented with a directed graph. *Causes* include implementation mistakes, conditions, preceding weaknesses and circumstances that bring about the fault. Some of the causes are nested hierarchically. The identifying or distinguishing *attributes* are the next general area.

Some assurance techniques or mitigation approaches may work for a fault with certain attributes, but not for the same general kind of fault that has other attributes. Each attribute is an enumeration of possible values. Lists of attributes also open the opportunity to more formally define and reason about them.

Note that the attributes describe an event, not the site in code that gives rise to the event.

We want to be able to forecast possible *consequences* of different kinds of faults. Knowing what consequences might occur allows risk estimation and determination of best mitigation strategies.

CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Description Summary: The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

Extended Description: A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold, or when a program attempts to put data in a memory area outside of the boundaries of a buffer. The simplest type of error, and the most common cause of buffer overflows, is the "classic" case in which the program copies the buffer without restricting how much is copied.

Common Consequences: Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. This can often be used to subvert any other security service. Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.

Fig. 3. Phrases in CWE-120 descriptions labeled according to ST phases. Blue is software faults. Yellow is a weakness. Green is resource or location. Red is consequences.

Finally, we describe the *sites* or locations in code where the bug might occur under circumstances indicated by the causes.

A site is a location in code where a weakness might be For instance, every buffer access in a C program is a site where buffer overflow might occur if the code is buggy. In other words, sites for a weakness are places that must be checked for that weakness. [9] The determination of sites depends only on local information. That is, global or flow-sensitive information is not needed to determine where sites are in code.

For example, the following code comes from Software Assurance Reference Dataset (SARD) [10] case 62 804. It has one site of writing to an array, `data[i] = ...`, which needs to be checked for a write-outside-array bug. There is also one site of reading from an array, `source[i]`, where the program might read outside the array if there is a bug.

```
for (i = 0; i < 10; i++) {
    data[i] = source[i];
}
```

In addition, the code has sites of possible uninitialized variable, every place that `i` is used, and a possible integer overflow site, `i++`. Notice that the assignment statement in the body of the loop has several sites.

This statement-level definition of site not always applies. When a C programmer uses the `strcpy` library function, it does not get enough information to check for a buffer overflow. Similarly the Structured Query Language (SQL) processor cannot determine that the programmer never intended queries like "name = Henry or 1=1" to be always true. The site is the last or lowest level of code execution outside library functions or utilities. This is the final chance the programmer had to avoid the fault. In other words, sites of a bug are places in the code that should be checked for that class of bug.

Following are one section for each of three classes of bugs from our BF: buffer overflows, injections and control of

frequency of interactions. In each section, we give a definition of the class and our taxonomy, including related sites. Following that we provide examples and related classes from other collections, such as CWEs and SFPs.

IV. Buffer Overflow Class – **BOF**

A. *Definition*

We define Buffer Overflow (BOF) as:

> *The software can access through an array a memory location that is outside the boundaries of that array.*

Often referred to as a "buffer," an array is a contiguously allocated set of objects [11], called elements. An array has a definite size—a definite number of objects are allocated to it. The elements are of same data type and are accessed by integer subscripts.

If the software can utilize the array name (more generally, array handle or pointer to array elements) to access any memory other than the allocated objects, it falls into this class.

B. *Taxonomy*

Fig. 4 depicts BOF causes, attributes and consequences.

The graph of causes for BOF shows that there are only two proximate causes of buffer overflows: amount of data exceeds the array size or there is a wrong index or pointer. Those two causes have preceding causes that may lead to them. Note that "Data Exceeds Array" could be the result of an "Array Too Small" or result of "Too Much Data." The former sub-cause means that an incorrectly small array has been allocated and because of that the destination is too small. The latter sub-cause means that incorrectly large amount of data has been accessed and because of that the data is too big.

The attributes of BOF are:

Access – Read, Write.

Boundary – Below, Above. This indicates which end of the array is violated. Synonyms for boundary are side or bound. The terms before, under or lower may be used instead of below. The terms after, over or upper may be used instead of above. Outside indicates that the boundary is unknown or it doesn't matter.

Location – Heap, Stack. This indicates what part of memory the array is allocated in. It may matter since violations in the stack may affect program execution flow, while violations in the heap typically only affect data values. Other architectures may have other locations that are significant. For instance, Intel architecture also has Bss, Data and Code (text).

Magnitude – Small, Moderate, Far. This is how far outside the boundary the violation extends. Small means just barely outside, e.g. one to a few bytes beyond the end, moderate is something like eight bytes to dozens, and far is hundreds, thousands or more.

These distinctions in the magnitude attribute are important because some violation detection techniques or mitigation techniques, such as canaries or allocating a little extra space, are only useful if the magnitude is small.

Data Size – Little, Some, Huge. This is how much data is read or written beyond the boundary. Like in magnitude, these distinctions are important in some cases. For instance, Heartbleed [12] would not have been a severe problem if it just exfiltrated a little data. The fact that it may exfiltrate a huge amount of data greatly increases the chance that very important information will be leaked.

Reach – Continuous, Discrete. This indicates whether the access violation was preceded by consecutive access of elements starting within the array (continuous) or just an access outside of the array (discrete). Typically string accesses or array copies handle a continuous set of array elements, while a vagrant array index only reads or writes one element.

Note that any of the attributes may be "any," "don't care" or "unknown." For instance, strict bounds checking is equally effective regardless of the location, magnitude, data size or reach of the violation. Keeping return addresses in a separate stack helps prevent problems occurring from write accesses when the array location is the stack.

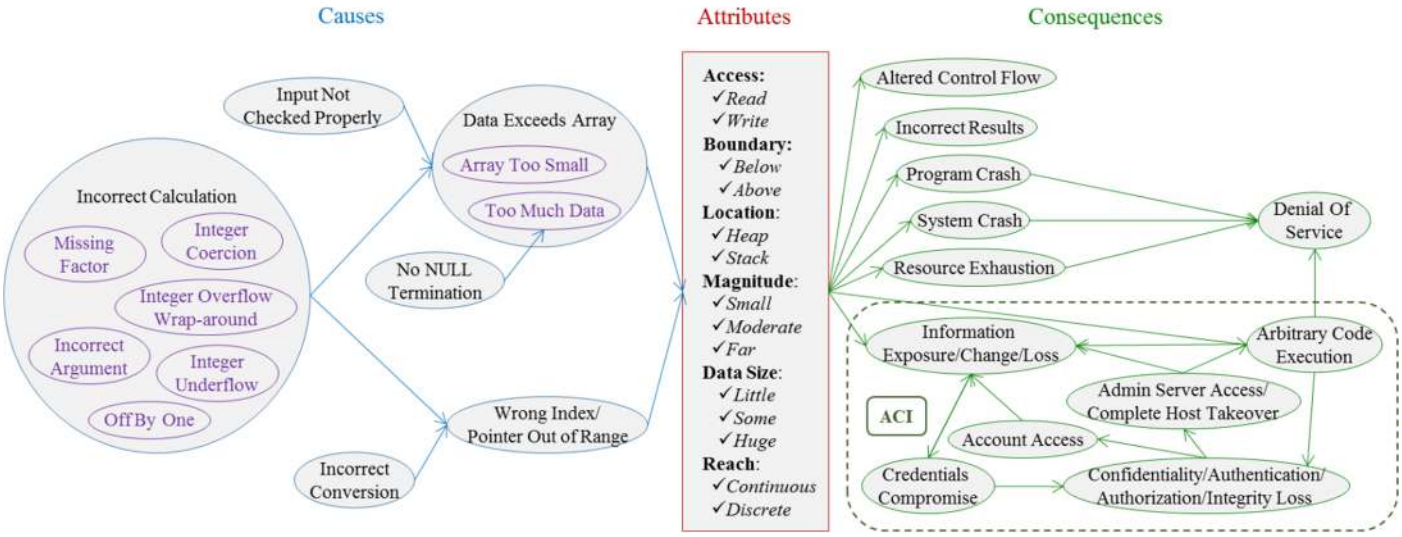

Fig. 4. The Buffer Overflow (BOF) class represented as causes, attributes and consequences. (The ACI cluster is the same in all classes.)

The values for the access, boundary, location, magnitude, and reach attributes were listed earlier by Kratkiewicz [13], although they were discovered independently through our analysis of the buffer overflow CWEs. Some additional attributes from [13] that might be relevant to our BOF taxonomy are: Data Type, for instance int, float or Boolean, and Container, for instance, the array is in a struct or record.

Note that in the graph of consequences in Fig. 4, "Resource Exhaustion" refers to Memory and CPU.

In the C language, sites where a buffer overflow may occur are the use of `[ ]` or unary `*` operators with arrays. Sites also include the use of string library functions as `strcpy` or `strcat`.

C. Examples

1) CVE-2014-0160 – Heartbleed

This vulnerability is listed in [12] and discussed in [14]. Our BF description is:

Input not checked properly leads to too much data, where a huge number of bytes are read from the heap in a continuous reach after the array end, which may be exploited for exposure of information that had not been cleared.

2) CVE-2015-0235 – Ghost

This vulnerability is listed in [4] and discussed in [15]. Our BF description is:

Incorrect calculation, (specifically missing factor) leads to array too small, where a moderate number of bytes are written to the heap in a continuous reach after the array end, which may be exploited for arbitrary code execution, leading to denial of service.

3) CVE-2010-1773 – Chrome WebCore

This vulnerability is listed in [4] and discussed in [16, 17, 18, 19, 20, 21]. Our BF description is:

Incorrect calculation, (specifically off by one) leads to a wrong index, where a small number of bytes are read from the heap in a discrete reach before the array start, which may be exploited for information exposure, arbitrary code execution or program crash, leading to denial of service.

D. Related CWEs, SFP and ST

CWEs related to BOF are: CWE-119, 120, 121, 122, 123, 124, 125, 126, 127, 786, 787 and 788. The only related SFP cluster is SFP8 Faulty Buffer Access under Primary Cluster: Memory Access [22]. The corresponding ST is the Buffer Overflow Semantic Template [23].

V. Injection Class – **INJ**

A. Definition

We define Injection (INJ) as:

Due to input with language-specific special elements, the software can assemble a command string that is parsed into an invalid construct.

In other words, the command string is interpreted to have unintended commands, elements or other structures.

B. Taxonomy

Fig. 5 depicts INJ causes, attributes and consequences.

The attributes of INJ are:

Language – Database Query, Regular Expression, Command, Markup, Script. This indicates the language in which the command string is interpreted. Database query language could be SQL. Command language could be Bash. Markup language could be XML/Xpath. HTML Scripting language could be PHP, CGI.

Special Element – Query Elements, Header Separators, Scripting Elements, Format Parameters, Path Traversals,

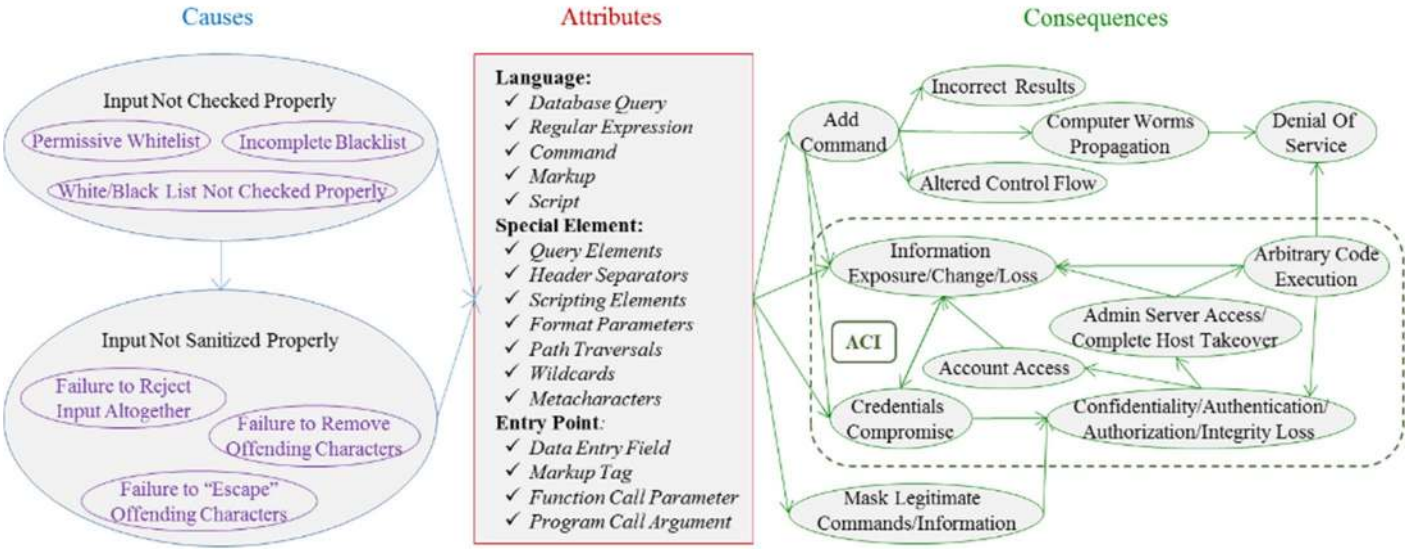

Fig. 5. The Injection (INJ) class represented as causes, attributes and consequences. (The ACI cluster is the same in all classes.)

Wildcards, Metacharacters. These could be assembled with other elements to form malicious structures such as queries, scripts and commands. Query elements are strings delimiters ‘ or “ or words such as ‘and’ or ‘or’. Header separators are carriage return/line feed. Scripting elements are < or > or &. Format parameters are such as %c or %n. Path traversals elements are .. or \. Metacharacters are back tick (`) or $ or &.

Entry Point – Data Entry Field, Scripting Tag, Markup Tag, Function Call Parameter, Procedure Call Argument. This indicates where the input came from.

Note that in the graph of consequences on Fig. 5, “Arbitrary Code Execution” concerns any instructions to the computer – compiled, interpreted by software, executed directly by hardware or combination.

Injection sites are typically not primitive operations in most languages. Sites are the library or utility functions that accept a command string for actions. In shell commands, command substitution is invoked with paired back quotes (`…`) or $(…). Command substitution executes a subshell, which opens the possibility of the string to be interpreted with all the richness of the command line interpreter.

C. Examples

1) CVE-2007-3572 – Yoggie Pico

This vulnerability is listed in [4] and discussed in [25]; special elements are discussed in [26]. Our BF description is:

Input not checked properly (specifically incomplete blacklist) allows shell command injection through the “param“ function parameter in a CGI script using Shell metacharacters (specifically back ticks `), which may be exploited to add command, leading to arbitrary code execution.

Note that adding a command through Ping to change the root password enables eventual complete host takeover.

2) CVE-2008-5817

This vulnerability is listed in [4] and discussed in [27, 28]. Our BF description is:

Input not checked properly or input not sanitized properly allows SQL injection through the “username“ & “password“ fields in a PHP script using query elements (specifically ' , or, and =), which may be exploited to mask legitimate SQL commands, leading to authentication compromise, admin server access and arbitrary SQL code execution.

3) CVE-2008-5734

This vulnerability is listed in [4] and discussed in [29, 30, 31]. Our BF description is:

Input not sanitized properly allows XSS web script or HTML injection through the IMG element of a generated HTML email, which may be exploited to add commands or for cookie-based authentication credentials compromise, leading to arbitrary code execution.

D. Related CWEs, SFPs and ST

CWEs related to INJ are CWE-74, 75, 77, 78, 80, 85, 87, 88, 89, 90, 91, 93, 94, 243, 564, 619, 643 and 652. Related SFPs are SFP24 and SFP27 under Primary Cluster: Tainted Input, and SFP17 under Primary Cluster: Path Resolution [22]. The corresponding ST is the Injection Semantic Template [32].

VI. Control of Interaction Frequency Class – **CIF**

A. Definition

We define Control of Interaction Frequency (CIF) as:

The software does not properly limit the number of repeating interactions per specified unit.

In physics, frequency is the number of occurrences of a repeating event per unit time [24]. Interactions in software could be also per event or per user.

B. Taxonomy

Fig. 6 depicts CIF causes, attributes and consequences.

The attributes of CIF are:

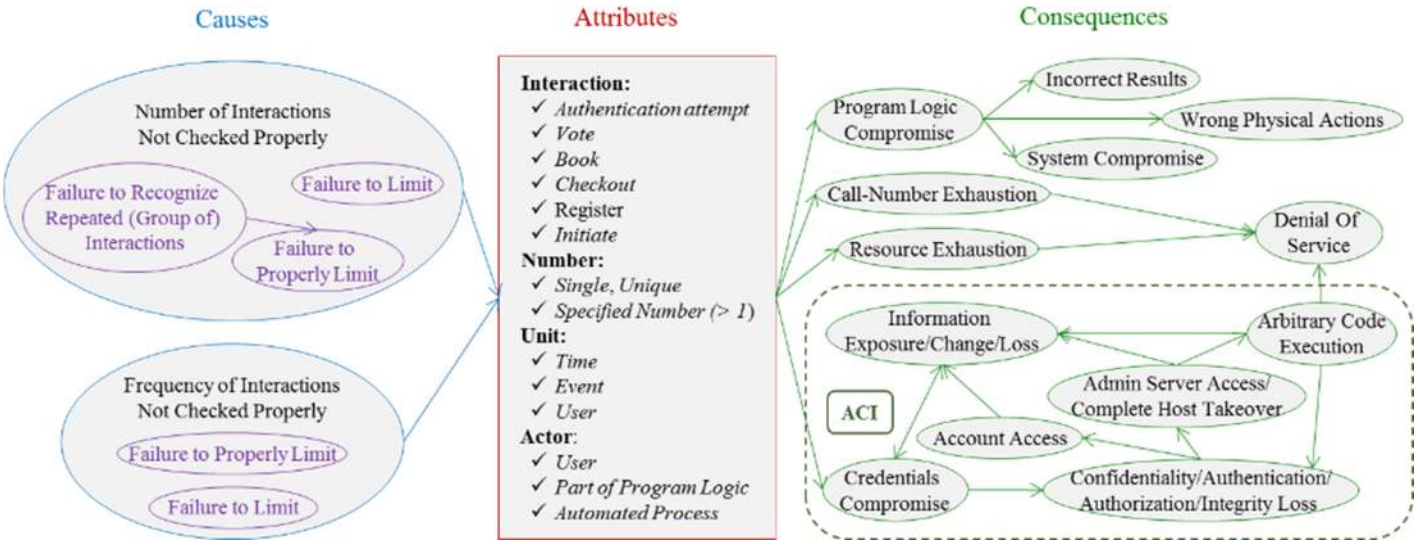

Fig. 6. The class Control of Interaction Frequency (CIF) represented as causes, attributes and consequences. (The ACI cluster is the same in all classes.)

Interaction – Authentication Attempt, Book, Checkout, Register, Initiate. This indicates the type of interactions to be controlled. Voting could be related to election, census, survey, referendum and ballot. Booking could be of tickets, hotel rooms or rental cars. Checkout could be of library books, hotel rooms or rental cars. Register could be for computer games. Initiate could be for message exchange.

Number – Single, Unique; Specified Number (> 1). This indicates the maximum number of occurrences allowed.

Unit – Time Interval, Event, User. This indicates the specific unit per which the number of occurrences is controlled. Time Interval could be in seconds, in days, etc. Event could be election, authentication, on-line transaction to move funds, etc.

Authentication event is the sequence of authentication attempts arriving at a particular server, possibly with the same partial credential, from any source, that terminates by successful authentication or by blocking.

Actor – User, Part of Program Logic, Automated Process. This indicates who/what is performing the repeating interactions. User could be authenticated user, attacker. Part of program logic could be message exchange. Automated process could be virus, bot.

Note that in the graph of consequences in Fig. 6, "Credentials" concerns username or password, smart card and personal identification number (PIN), retina, iris, fingerprint, etc. "Resource Exhaustion" concerns memory, CPU or granted licenses.

Our taxonomy makes it abundantly clear that CIF is a "metaclass" in some senses. External policies must define for each system or application what constitutes an interaction, how many interactions should be allowed, and the unit. Each policy, then, defines a different class of CIF concerns.

Since the concept of interaction is so broad and high level, compared to most programming languages, no general description of what is a site is feasible. Each system or application must define its own concept of interaction. An interaction must then be mapped to some code that controls or authorizes said interactions. More importantly, since a failure may be the total lack of code to recognize and control frequency of interaction, there is often no particular line or even block of code that can be pointed out as missing the control code. An entire path may be indicated from the beginning of an interaction event, that is, an outside agent indicates desire to start an interaction, to the final chance in execution flow that code may refuse to authorize the event.

C. Examples

1) CVE-2002-0628

This vulnerability is listed in [4]. Our BF description is:

Failure to limit to a specified number the authentication attempts per authentication event by same or different user(s) may be exploited for credentials compromise (username or password) via brute force.

2) CVE-2002-1876

This vulnerability is listed in [4]. Our BF description is:

Failure to recognize repeated interactions that are rapid initiations of message exchange requests from authenticated users, leads to failure to properly limit them to a specified number per specified time interval, which may be exploited for resource exhaustion (consumption of all granted licenses) leading to denial of service.

3) CVE-2002-1018

This vulnerability is listed in [4]. Our BF description is:

Failure to limit the checkouts of a book to a single one per user may be exploited for resource exhaustion leading to denial of service.

D. Related CWEs and SFP

CWEs related to CIF are CWE-799, 307 and 837. The related SFP cluster is SFP34 Unrestricted Authentication under the Primary Cluster: Authentication [22].

VII. Conclusions

A. Summary

We have shown a superior, unified approach. The presented Bugs Framework (BF) allows accurate, precise and unambiguous expression of software bugs or vulnerabilities. It can also be used to clearly explain the applicability and utility of different software quality or assurance techniques or approaches, which is demonstrated in the discussion of the magnitude and data size attributes of BOF.

This approach is a factoring and restructuring of information contained in CWEs, SFPs and STs, and thus benefits from the community's experience with their use. Instead of trying to match weakness classes that tools find to CWEs, usually far over- or under-generalizing, the BF can describe tool classes much more accurately, precisely and succinctly. Table 1 shows how this refinement approach allows clearer and more succinct descriptions. The BF consists of (1) causes arranged in a directed graph, (2) attributes of a software fault, (3) possible consequences of the fault, also in a directed graph and (4) possible sites in code, that is, locations that must be reviewed for possible faults. Causes and consequences may be hierarchical, too. For instance, "Data Exceeds Array" in BOF is either "Array Too Small" or "Too Much Data." "Input Not Checked Properly" in the INJ class is either "Permissive Whitelist" or "Incomplete Blacklist."

B. Benefits

With our BF practitioners and researchers can more accurately, precisely and clearly describe problems in software, discuss the classes of bugs that tools report or explain what vulnerabilities the proposed techniques prevent. Instead of adding more and more CWEs for every slight variant, types of weaknesses can be categorized unambiguously, allowing similarities and differences to be easily explored and examined. We believe that as CWEs migrate to using this kind of taxonomy, they will be easier to comprehend and avoid.

Those concerned with software quality, the reliability of programs and digital systems, or cybersecurity will be able to make more rapid progress now that they can more clearly label the results of errors in software. Those responsible for designing, operating and maintaining computer complexes can communicate with more exactness about threats, attacks, patches and exposures.

C. Future Work

Although we demonstrate this approach on three disparate classes of weaknesses, much work remains. More examples, such as CVEs and tool classes, need to be expressed using this scheme to bring out facets that were overlooked or find better ways of organizing the information. Consequences need to be examined across all classes to better understand how, say, adding commands can lead to whole host takeover, regardless of the weakness allowing the addition. Chains of causes should be researched similarly. A concerted investigation of chains through particular attributes, drawing on existing work, should help clarify the relations between them. We also need to refactor many other bugs classes, which will turn up commonalities. We are building a web site at https://samate.nist.gov/BF/ which will have the latest information, such as guide books for classes.

As our BF covers more classes, existing taxonomies, like CWE, can start explaining their current entries with concise forms of our descriptions. Bug trackers can be enhanced to allow BF descriptions to be given. Many tool makers, such as static analyzers and bug trackers, already use CWEs. In the future, this use of CWEs can evolve to integrate BF into bug descriptions. For instance, a software assurance tool maker can find a CWE similar to the class of bugs that their tool can find and start with its BF description. The tool maker then can refine the BF description, changing enumerations of attributes until it matches their tool's class.

REFERENCES

[1] The MITRE Corporation. Common Weakness Enumeration (CWE). http://cwe.mitre.org.

[2] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws, and examples. ACM Computing Surveys. vol. 26. no. 3. pp. 211–254. September 1994.

[3] W. Thomson, Baron Kelvin. Electrical units of measurement. Popular Lectures and Addresses. MacMillan. 1889. vol. 1. p. 73. A Lecture delivered at the Institution of Civil Engineers. May 3. 1883.

[4] The MITRE Corporation. Common Vulnerabilities and Exposures or (CVE). https://cve.mitre.org.

[5] The MITRE Corporation. Common Weakness Enumeration. CWE 121. https://cwe.mitre.org/data/definitions/ 121.html.

[6] N. Mansourov and D. Campara. System Assurance: Beyond Detecting Vulnerabilities. Morgan Kaufmann. 2010. pp. 176–188.

[7] Y. Wu, R. A. Gandhi, and H. Siy. Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories. Proc. 2010 ICSE Workshop on Software Engineering for Secure Systems. ser. SESS '10. New York, NY: ACM. 2010. pp. 22–28. http://doi.acm.org/10.1145/1809100.1809104.

[8] Y. Wu., I. Bojanova, and Y. Yaacov. They Know Your Weaknesses – Do You?: Reintroducing Common Weakness Enumeration. *CrossTalk* (The journal of Defense Software Engineering). Sept-Oct 2015. http://static1.1.sqspcdn.com/static/f/702523/26523304/1441780 301827/201509-Wu.pdf.

[9] P. E. Black and A. Ribeiro. SATE V Ockham Sound Analysis Criteria. National Institute of Standards and Technology (NIST). NIST IR 8113. March 2016. http://dx.doi.org/10.6028/NIST.IR.8113.

[10] Software Assurance Reference Dataset (SARD). https://samate.nist. gov/SARD.

[11] ISO/IEC 9899:2011 programming languages - C, Committee Draft— April 12, 2011 N1570. ISO/IEC Joint Technical Committee JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces. Working Group WG 14 – C. Tech. Rep. 2011.

[12] The MITRE Corporation. CVE-2014-0160. https://cve.mitre. org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[13] K. Kratkiewicz. Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Master's thesis. Harvard University, Cambridge, MA. March 2005. https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/KratkiewiczThesis.pdf.

[14] S. Cassidy. Diagnosis of the OpenSSL Heartbleed Bug. https:// www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html.

[15] Openwall. Qualys Security Advisory CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow. http://www.openwall. com/lists/oss-security/2015/01/27/9.

[16] R. Gandhi. Buffer Overflow Semantic Template: CVE-2010-1773. http://faculty.ist.unomaha.edu/rgandhi/st/CVE-2010-1773.pdf.

[17] The MITRE Corporation. CVE-2010-2304. http://cve.mitre.org/ cgibin/cvename.cgi?name=CVE-2010-2304.

[18] Debian Bug report logs - #586547. Webkit: CVE-2010-2304 memory corruption in rendering of list markers. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=586547.

[19] Chromium. Diff of /branches/WebKit/375/WebCore/rendering/ RenderListMarker.cpp. http://src.chromium.org/viewvc/chrome/branches/WebKit/375/WebCore/rendering/RenderListMarker.cpp?r1=48100&r2=48099.

[20] Chromium. Contents of /branches/WebKit/375/WebCore/rendering/ RenderListMarker.cpp. http://src.chromium.org/viewvc/chrome/branches/WebKit/375/WebCore/rendering/RenderListMarker.cpp?annotate=48100#l104.

[21] Red Hat Bugzilla – Bug 596500 CVE-2010-1773 WebKit: off-by-one memory read out of bounds vulnerability in handling of HTML lists. https://bugzilla.redhat.com/show_bug.cgi?id=596500.

[22] B. A. Calloni, D. Campara, and N. Mansourov. White Box Definitions of Software Fault Patterns. Final Report. Lockheed Martin Corporation and KDM Analytics, Inc. 2011.

[23] R. Gandhi, H. Siy, and Y. Wu. Buffer Overflow Semantic Template. http://faculty.ist.unomaha.edu/rgandhi/st/buffer overflowtemplate.pdf.

[24] Wikipedia. Frecuency. https://en.wikipedia.org/ wiki/Frequency.

[25] Neohapsis. Yoggie Pico Pro Remote Code Execution. http://archives.neohapsis.com/archives/fulldisclosure/2007-07/0020.html.

[26] The MITRE Corporation. Common Weakness Enumeration. CWE 78. https://cwe.mitre.org/data/definitions/78.html.

[27] Cxsecurity. WebClassifieds 2005 (Auth Bypass) SQL Injection Vulnerability. http://cxsecurity.com/issue/WLB-2009010117.

[28] Mozilla Developer Network. SQL Injections. https://developer. mozilla.org/en-US/docs/Glossary/SQL_Injection.

[29] Secunia. Merak Mail Server Web Mail "IMG" HTML Tag Script Insertion. http://secunia.com/advisories/32770.

[30] N. Vijatov. Vulnerability in Merak Mail. http://secunia.com/ advisories/32770.

[31] Securyty Focus. Merak Mail Server and Webmail Email Message HTML Injection Vulnerability. http://www.securityfocus.com/bid/ 32969/info.

[32] R. Gandhi, H. Siy, and Y. Wu. Injection Semantic Template. http://faculty.ist.unomaha.edu/rgandhi/st/injec tiontemplate.pdf.