Lightweight Formal Models of Software Weaknesses

Robin Gandhi, Harvey Siy, Yan Wu
College of Information Science and Technology
University of Nebraska at Omaha, USA
{rgandhi, hsiy, ywu}@unomaha.edu

***Abstract*—Many vulnerabilities in today's software products are rehashes of past vulnerabilities. Such rehashes could be a result of software complexity that masks inadvertent loopholes in design and implementation, developer ignorance/disregard for security issues, or use of software in contexts not anticipated for the original specification. While weaknesses and exposures in code are vendor, language, or environment specific, to understand them we need better descriptions that identify their precise characteristics in an unambiguous representation. In this paper, we present a methodology to develop precise and accurate descriptions of common software weaknesses through lightweight formal modeling using Alloy. Natural language descriptions of software weaknesses used for formalization are based on the community developed Common Weakness Enumerations (CWE).**

***Index Terms*—Software weakness, Alloy modeling, CWE.**

I. Introduction

Precise and accurate weakness descriptions are needed as one element to speed the work of detecting and preventing those weaknesses in software. While precision, accuracy and automation may be achieved by formal representations, such formality must be balanced with accessibility for review by software developers that engage in a software implementation effort, which is still for the most part a manual activity. Thus, an important prelude to preventing, mitigating, or detecting weaknesses in software and systems is to have clear, unambiguous, widely accepted definitions of such weaknesses.

The Common Weakness Enumeration (CWE) provides a unified and measureable set of software weaknesses for use in software assurance activities [6]. CWE is a community driven and continuously evolving taxonomy of software weaknesses. The CWE vision is to enable a more effective discussion, description, selection, and use of software security tools and services that can find weaknesses in source code and operational systems as well as better understanding and management of software weaknesses related to architecture and design. The CWE is often compared to a "Kitchen Sink", although in a good way, as it aggregates weakness categories from many different vulnerability taxonomies, software technologies and products, and categorization perspectives. While the CWE is comprehensive, using its highly tangled web of weakness categories to study vulnerabilities for a particular software project is a daunting task for stakeholders in the software development lifecycle.

Much work has been done on the CWE but there are still ambiguities, and perhaps errors, in its various weakness definitions. Consider the following questions that might occur to someone learning about software weaknesses.

- Is a buffer overflow (CWE-119) the same as a stack overflow (CWE-121) or an unbounded transfer (CWE-120) or is one just a refinement of another?
- If an integer overflow (CWE-190) leads to memory bounds violation (CWE-119), which weakness is it? Is it both or is there some other relation between them?

A quest to develop precise definitions of weakness could systematically raise these questions about ambiguity, lead to consolidation or differentiation among CWEs, all while providing uniformity in their interpretations. Several approaches exist to formally specify software engineering artifacts (e.g. requirements, designs, programs, etc.). However, software weaknesses are rarely modeled formally due to their abstract descriptions. We envision that a formally specified collection of weaknesses would suggest properties that should not be exhibited in a software specification as well as a tightly constrained set of permitted behaviors.

II. Why Lightweight Formal Modeling Using Alloy?

Given the declarative and abstract nature of security weaknesses; it is highly desirable to understand the concrete circumstances under which a design permits them. Therefore, in our approach we codify community based weakness definitions as unambiguous, readable and reusable declarative specifications that are fully executable in a bounded scope. We parameterize the natural language expression of weakness definitions into security relevant concepts and express the necessary and sufficient conditions in a lightweight formal modeling language. A formal specification is highly desirable as it enables a fully or semi-automated analysis of system behavior. Particularly for analysis of security weaknesses, in contrast to model checking, we find that a model finding problem [5] is a more appropriate approach that addresses these interesting set of questions:

Counterexample model finding: With a certain weakness what unwanted behaviors can the software exhibit? In other words, what unwanted system models exist that exemplify the specified security weakness? How does an under-constrained software violate security expectation?

Valid model finding: Can a valid instance of software behavior exist if the software weakness is designed out? In other words, does the system permit acceptable functional behavior with the security weakness accounted for?

To support these model-finding activities we have chosen Alloy [5]. Alloy is a lightweight model finder with a fully automatic analysis that provides immediate feedback. It is based on relational logic that combines first order logic with the operators of relational calculus. All structures in this logic are built from immutable and un-interpreted atoms, and their relationships. The Alloy language is declarative with the following key elements:

1. Signature declarations: (keyword **sig**) that declare a set of atoms and a set of fields that represent relations. Signature declarations define the static structure of a model.
2. Constraints: (keywords **fact**, **pred**, **fun**, **assert**) "Facts" represent constraints that are always true. "Predicates" represent a constraint that can be reused in facts, other predicates or used to simulate model instances that preserve the constraint. "Functions" are also reusable constraints like predicates but have a return type other than Boolean. "Assertions" record constraints that are expected to hold, but interact with the model only when required (in contrast to facts that are "always on").
3. Commands: (keywords **run**, **check**) are instructions to the analyzer to perform particular analyses. "Run" simulates the model with respect to predicates or functions, where as "Check" explores counterexamples for assertions.

Alloy analysis is done via a satisfiability solver; given constraints on variables, it finds a set of instances to the variables that satisfies the constraints. Satisfiability solvers are generally undecidable, but Alloy will exhaustively search for a satisfying model instance within a bounded scope. Because of a bounded scope if no model instance is found, there is a possibility that it may exist in a larger scope.

Alloy's model finding exercise of looking for refutations of an assertion closely mimics the activity of a malicious and dedicated adversary that seeks to undermine the security of a design by exploiting weakness (under-constrained behavior). Just as refutation-based analysis requires discovering only one counterexample to invalidate an assertion, an attacker only needs to discover one possible instance of unexpected, but allowed, software behavior to undermine or bypass defenses.

III. Methodology

We propose the following modeling approach to develop lightweight formalization of CWE definitions:

STEP 1: Preparing CWEs

An initial activity is to study the textual definitions of the weakness and identify the central concepts involved. In particular, we focus on the answers to the following questions in a CWE definition:

1. WEAKNESS: What are the discernable conditions necessary to establish the existence of the weakness? (They explain "What" conditions signify a weakness.)
2. FAULT: What are the software faults that can precede the weakness conditions? (Software faults, in the form of allowed behaviors, are precursors that bring about the weakness conditions. They explain "How" weakness conditions occur.)
3. RESOURCE/LOCATION: What are the resources and/or locations where the weakness conditions commonly occur?
4. CONSEQUENCE: What are the consequences i.e. what are the failure conditions that the weakness conditions can lead to?

These four questions have also led to the development of *semantic templates* to assist programmers in the study of software vulnerabilities. The development of such semantic templates and related experiments are described elsewhere [4][7]. Here we focus on the lightweight formal description of these concepts identified from CWE textual definitions.

The CWE is a collection of interrelated weakness definitions. Therefore, for a CWE that is the target of the formalization activities, we navigate to its parent CWEs and repeat this step for their definitions. This activity ensures that all domain concepts and related constraints implicitly inherited from parent CWEs are considered during the formalization.

STEP 2: Identification of Domain Concepts and Relationships

The next step is to study the textual descriptions of the software fault, weakness, location/resource and consequences to identify the domain concepts involved, much like object identification in object-oriented analysis [2]. In the process, the scope of the formal model is defined. The identified concepts form the basis for authoring Alloy signatures.

Next, we identify which concepts are related and describe their relationships, using object modeling techniques. Relationships are named and properties of those relationships (e.g., cardinality, etc.) are determined. These relationships become fields within their respective Alloy signatures. In this step, it also becomes necessary to identify domain constraints as certain facts about the concepts or their relationships (e.g. multiplicities, functional, inverse, etc.).

STEP 3: Modeling Software Operational Behavior

This step defines Alloy predicates that models the aspect of software operational behavior upon which the weakness can arise. A key challenge is determining the appropriate level of abstraction to use in defining the behavior. The specification should not be too high level that it no longer enables meaningful analysis. It should also not be too low-level as this would be better served by code examples.

In case studies presented later in the paper, we strive for the level of abstraction to match as closely to the CWE textual definition. CWE annotation of a weakness as class (abstract), base (has details about detection and/or prevention) or variant (limited to a specific language or technology) further helps the effort. Most base weaknesses tend to be abstract, so operations are defined on software elements that are largely left unspecified. On the other hand, weakness variants are more low level, in which case, operations more closely emulate the data structures. Depending on the weakness being modeled, the behavior can also be modeled as static (stateless) or dynamic (stateful). Stateless operations treat each instance of the operation as independent of prior operations while stateful operations depend on certain prior operations involving the same signatures. While Alloy does not have built-in support for time or mutable state, these are explicitly included in a model.

Upon modeling the predicates, simulate them by running them using Alloy Analyzer. The discovered model instance may or may not exhibit the weakness condition.

STEP 4: Checking for Violations of the Desired Behavior

Once the operational behavior has been modeled, the desired behavior is declaratively specified in terms of the desired post-condition resulting from executing the operation specified using Alloy predicates. Note that predicates are "reusable constraints" in the sense that they are only invoked upon request. In this step, the predicates are invoked using Alloy assertions that declaratively specify the desired post-conditions. The assertions can then be checked automatically within a bounded scope, using the Alloy Analyzer. If found a model instance is discovered it acts as a counter-example for the stated assertion. Violations of the assertions can be considered as manifestations of the weakness and represent software behavior that can lead to exploits.

STEP 5: Enforcing the Desired Behavior

In this step, the goal is to define invariant properties on the operation that prevent the weakness from occurring. This provides a validation step for the modeling process. The invariant is expressed as an Alloy fact. By running the Alloy Analyzer with these invariants, it is expected that no assertion violations should occur (i.e. no counterexamples discovered in the given search scope upon checking the assertion in Alloy Analyzer), while simulating the predicates (i.e. running the predicate in Alloy Analyzer) still produces valid and expected system model instances. Note that only the explicitly expressed assertions and predicates are analyzed for consistency.

IV. Case Study

We illustrate our modeling approach on CWE-119: *Improper Restriction of Operations within the Bounds of a Memory Buffer*, which is described as follows:

Description Summary: *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.*

STEP 1: Prepare CWEs

CWE-119 is the most abstract description of a buffer overflow weakness. It is also a class weakness (abstract). Using the four questions in this step the following aspects are identified from the CWE textual description.

- WEAKNESS-0: Improper restriction of operations within the bounds of a memory buffer
- WEAKNESS-1: Software can read from a memory location that is outside of the intended boundary of the buffer
- WEAKNESS-2: Software can write to a memory location that is outside of the intended boundary of the buffer
- RESOURCE-1: Memory Buffer

Note that not all four aspects are identified in most CWE descriptions. Some CWEs descriptions are resource focused, where as others are fault and weakness focused.

STEP 2: Identification of Domain Concepts and Relationships

The following domain concepts are extracted:

- Software, Operations (e.g. read and write), Memory Buffer, and Memory Locations

Furthermore, these concepts are related as follows:

- Buffers consists of memory locations
- Locations are either part of a buffer or not
- Software performs operations on a buffer at certain memory locations

With this information, here is an initial Alloy specification:

```
// Signature: Buffer with a relation contains
// that identifies the set of locations that
// are contained in the buffer
sig Buffer {
     contains: set Location
}
// Signature: A Location is part of zero or
// more Buffers. (When no multiplicity is
// specified for a field the default is one.)
sig Location{
     partof: set Buffer
}
// partof relation is the inverse of contains
fact {
  partof = ~contains
}
// Signature: Software with a relation.
// performsOperation relation is a three way
// mapping associating Software, Buffer, and
// Location. It contains the tuple s->b->l
// when Software s performs an operation on
// Buffer b at location l. The multiplicities
// indicate that many-to-many relationship
// can exist between members of Buffer and
// Location for a member of Software.
sig Software{
  performsOperation:
     Buffer set -> set Location
}
```

Alloy visualizes the metamodel as shown in Figure 1.

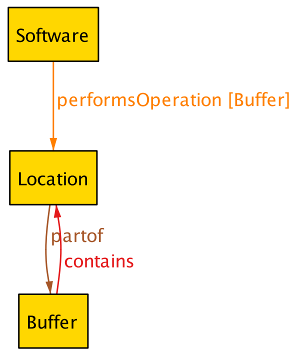


Figure 1. CWE-119 - Alloy metamodel.

STEP 3: Modeling Software Operational Behavior

Next, predicates are added. To formally specify the description of CWE-119, we first simulate a software operation on a buffer in an Alloy specification. The predicate **"operate"** models the execution of a buffer read or write as a simple stateless operation that is performed on buffer named *b* at a location *l*.

```
// Predicate: Operate that captures a software
// operation on a buffer at a location
// Running this predicate helps confirm
// that the model produces valid instances
// In Alloy, "in" can be read as "subset of".
pred operate
[s: Software, b: Buffer, l: Location] {
   b->l in s.performsOperation
}
// Run predicate in the specified scope
run operate for
1 Buffer, 3 Location, 1 Software
```

The execution ("Run") of the predicate instantiates the signatures involved in the predicate that satisfies the structures formed by the defined relationships. Note that the operation is bound for using one member of buffer, three members of locations and one member of software. We wanted to study the effect on a particular buffer, hence bounding the operation to one buffer. Similarly for software bounding it here means we are only looking at the effect of executing one process or thread. The bound of 3 locations enables the analyzer to examine various instances with 3 locations, some associated with buffer and others which are not. The "small scope hypothesis" of Alloy states that if an assertion is invalid then it probably has a small counterexample. So it recommended to start with a small scope and gradually expand it if a countermeasure is suspected to exist in a larger scope. Figure 2 below shows a model instance that results upon execution of the operate predicate. The simulation highlights *Software* operations performed on *Locations* in the context of a single *Buffer*. While operation on Location0 is expected, the operation on Location1 is outside the bounds of the intended *Buffer*. Several other instances can be found by browsing the generated instances of the **operate** predicate.

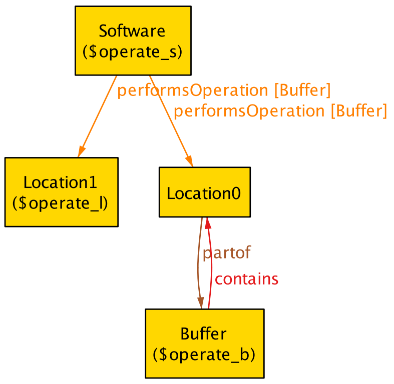


Figure 2. Simulation of the Operate Predicate.

STEP 4: Specifying the Desired Behavior

Instead of manually combing through all instances resulting from the operate predicate, we use the analyzer to identify all such violations by providing an assertion to check against.

```
// Assertion: All software operations on a
// Buffer are restricted within the bounds
// of the buffer.
assert noBufferOverflow{
   all s: Software, b:Buffer, l:Location|
       operate[s,b,l] implies l in b.contains
}
// Check assertion in the specified scope
check noBufferOverflow for
1 Buffer, 3 Location, 1 Software
```

By checking the assertion ("Check") Alloy returns the model instances that are counter-examples, in this case, all instances where there is a buffer overflow violation. Figure 3 shows one such violation as reported by the assertion check.

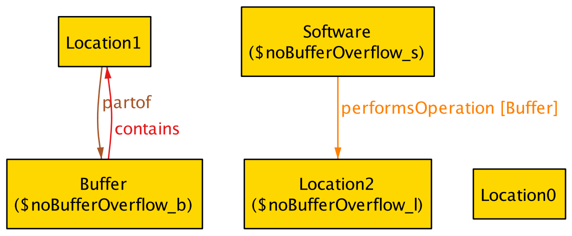

Location1
Software ($noBufferOverflow_s)
partof
contains
performsOperation [Buffer]
Buffer ($noBufferOverflow_b)
Location2 ($noBufferOverflow_l)
Location0

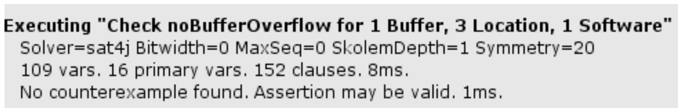
Figure 3. Buffer Access Violation as Counter-Example of Assertion

STEP 5: Enforcing Desired Behavior

To avoid the undesired behavior of writing outside the buffer, we can further constrain the model. This constraint is introduced as an invariant.

```
// Fact: Software operations on a Buffer are
// restricted to the locations in the bounds
// of that Buffer
fact limitBufferOverflow {
  all s: Software, b:Buffer|
        s.performsOperation[b] in b.contains
}
```

After introducing this fact, the simulation only produces model instances where CWE-119 weakness does not occur. Since the model is appropriately constrained, no counterexamples result on re-checking the "noBufferOverflow" assertion (Figure 4). With this fact introduced, the "**operate**" predicate still produces valid model instances.



```
Executing "Check noBufferOverflow for 1 Buffer, 3 Location, 1 Software"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  109 vars. 16 primary vars. 152 clauses. 8ms.
  No counterexample found. Assertion may be valid. 1ms.
```

Figure 4. Results After Desired Behavior is Enforced.

V. Case Study: CWE-787

To illustrate the reuse and extension of Alloy specifications for more specific weaknesses, we modeled CWE-787, whose parent is CWE-119. CWE-787: "Out-of-bounds Write" is [6]:

Description Summary: *The software writes data past the end, or before the beginning, of the intended buffer.*

STEP 1: Prepare CWEs

CWE-787 is a specialized version of CWE-119, thus it inherits the aspects identified for CWE-119 in the previous section. CWE-787 is classified as a base weakness i.e. it has details about detection and/or prevention. The following aspects are identified from the CWE-787 textual description.

- WEAKNESS-3: Out-of-bounds Write
- WEAKNESS-4: Software writes data past the end of the intended buffer
- WEAKNESS-5: Software writes data before the beginning of the intended buffer

Notice that WEAKNESS-3 is equivalent to WEAKNESS-2 identified for CWE-119. However, WEAKNESS-4 and 5 are more specific to WEAKNESS-2 as they specify the precise locations, i.e. at the start and end of a buffer.

STEP 2: Identification of Domain Concepts and Relationships

The following domain concepts, in addition to those in CWE-119, are extracted from the weakness description:

- Write operation, Memory Buffer Start Location, Memory Buffer End Location

Furthermore, an additional relationship is identified:

- Buffers consist of a set of memory locations that begin with a start location and terminate in an end location.

With this information, here is how we extend the CWE-119 Alloy specification for CWE-787. Due to limited space, we only include comments where necessary:

```
sig Buffer {
   contains: set Location,
   // Multiplicity: one indicates exactly one
   start: one StartLocation,
   end: one EndLocation
}
sig Location {
   partof: set Buffer
   // Multiplicity: lone indicates zero or one.
   previousLocation: lone Location,
   nextLocation: lone Location,
   //  Int,  a  predefined  signature  represents
   // a set of integer atoms
   position: lone Int
}
// There exists a StartLocation
one sig StartLocation extends Location {}
// A compact way to state facts
// related to the StartLocation
{  no previousLocation
   position = 1 }
// There exists an EndLocation
one sig EndLocation extends Location {}
// A compact way to state facts
// related to the EndLocation
{  no nextLocation }
```

The following domain facts are iteratively derived to rule out inconsistent buffer data structures obtained upon simulation of the operate predicate.

```
fact{
// A Location pointed to by nextLocation from a
// Location has a previousLocation relationship
// with the latter
all l1: Location, l2: l1.nextLocation |
   l2.previousLocation = l1
// A Location pointed to by nextLocation from a
// Location has a position value that is
// greater that one
all l1: Location, l2: l1.nextLocation |
   l2.position = l1.position.plus[1]
// A Location always has a positive position
all l: Location | l.position > 0
// No two Locations have the same position
all l,l': Location |
   disj[l,l'] implies
      disj [l.position, l'.position]
// The operator "*" denotes reflexive
// transitive closure.
// A Location is part of a buffer only it can
// be reached from the startLocation by
// following the nextLocation relationship.
all l: Location, b: Buffer |
   l in b.start.*nextLocation iff
      l in b.contain
// Start and end locations are in the buffer.
all l: Location, b: Buffer |
   l in b.start implies l in b.contains
all l: Location, b: Buffer |
   l in b.end implies l in b.contains
}
```

STEPS 3-5: Modeling, Checking and Enforcing Desired Operational Behavior

The predicate `operate` and assertion `noBufferOverflow` from the specification for CWE-119 are used as is for CWE-787. If desired, the performsOperation relation can be renamed to performsWriteOperation. Since it does not affect the model structure we do not introduce the change here.

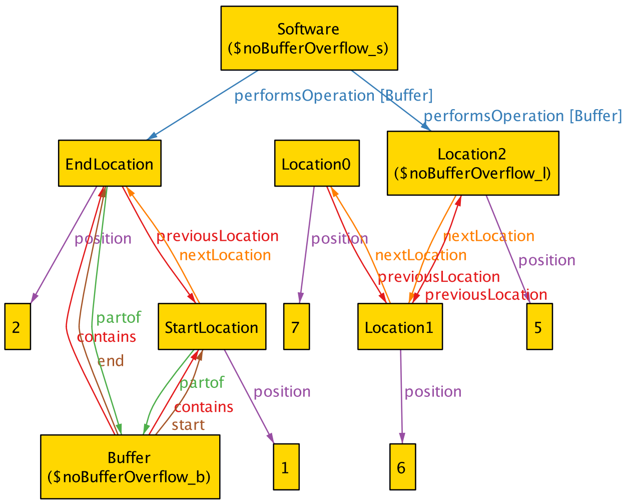


Figure 5: Counterexample for `noBufferOverlow` assertion

The counter-example that results upon checking the assertion is shown in Figure 5 where the *Software* writes past the

EndLocation at *position* 2 onto *Location2* at *position* 5. The scope of this check includes five *Locations*. Other counter-examples, possibly simpler, can be examined by asking alloy analyzer to report the "next" instance found. After re-introducing the fact `limitBufferOverflow`, from CWE-119, the simulation only produces model instances where CWE-787 weakness does not occur.

VI. Case study: CWE-121

CWE-787 is a parent to CWE-121, which is classified a variant weakness, i.e. it is limited to a specific language or technology. We also created an Alloy specification for CWE-121 to illustrate more details added on to a specialized version. CWE-121: "Stack-based Buffer Overflow" is described as [6]:

Description Summary: *A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).*

STEP 1: Prepare CWEs

CWE-121 inherits the aspects identified for CWE-787 as well as CWE-119 in the previous sections. Using the four questions identified before in this step the following aspects are identified from the CWE-121 textual description.

- WEAKNESS-6: Buffer overflow condition
- RESOURCE-2: Stack-based Buffer

Notice that WEAKNESS-6 is equivalent to WEAKNESS-2 identified for CWE-119. However, RESOURCE-2 is a more specific type of RESOURCE-1 identified in CWE-119.

STEP 2: Identification of Domain Concepts and Relationships

The following domain concept, in addition to those in CWE 119 and CWE-787, is extracted from the weakness description:

- Stack

Furthermore, an additional relationship is identified:

- Buffer is allocated on the Stack

With this information, here is how we extend the CWE-787 Alloy specification for CWE-121:

```
sig Buffer {
   contains: set Location,
   start: one StartLocation,
   end: one EndLocation,
   // Buffer is allocated to a Memory Area
   allocatedTo: one MemoryArea
}
// Stack, Heap, and Static are
// types of Memory Areas
abstract sig MemoryArea {}

sig Stack, Heap, Static extends MemoryArea {}

// Fact: Buffers are allocated on the Stack
fact {
   all b:Buffer |
       b.allocatedTo in Stack
}
```

With this revised structure, the predicate `operate`, assertion `noBufferOverflow`, and fact `limitBufferOverflow` are simply reused from the specification for CWE-787.

VII. Case study: CWE-307

To illustrate how the methodology would work on a completely different type of weakness, we developed a model for CWE-307: "Improper Restriction of Excessive Authentication Attempts," described as follows [6]:

Description Summary: *The software does not implement sufficient measures to prevent multiple failed authentication attempts within in a short time frame, making it more susceptible to brute force attacks.*

STEP 1: Prepare CWEs

While the description of CWE-307 is relatively brief, it inherits much from the descriptions of its parents. The complete ancestry is shown in Figure 6.

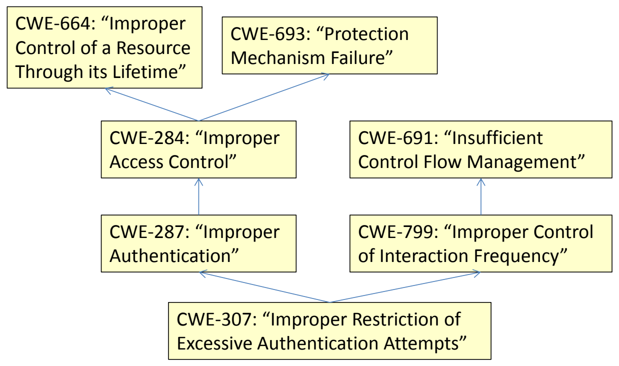

CWE-664: "Improper Control of a Resource Through its Lifetime"
CWE-693: "Protection Mechanism Failure"
CWE-284: "Improper Access Control"
CWE-691: "Insufficient Control Flow Management"
CWE-287: "Improper Authentication"
CWE-799: "Improper Control of Interaction Frequency"
CWE-307: "Improper Restriction of Excessive Authentication Attempts"

Figure 6: Ancestry of CWE-307

We have analyzed and developed Alloy models for each of these weaknesses. Due to space limitations, we only show here the relevant weaknesses and resources/locations identified from the parents and ancestors. With respect to CWE-307, the most relevant identified weaknesses and resources are:

- WEAKNESS-8: Failure to prove the identity of an actor (CWE-287)
- WEAKNESS-9: Failure to limit frequency of interactions within a given timeframe (CWE-799)
- WEAKNESS-10: Failure to limit number or frequency of authentication attempts
- RESOURCE-1: Resource (CWE-664)
- RESOURCE-2: Restricted Resource (CWE-284)
- RESOURCE-3: Actor (CWE-284)
- RESOURCE-4: Proof of Identity/Credential (CWE-287)
- RESOURCE-5: Time Frame (CWE-799)
- LOCATION-1: Protection Mechanism (CWE-693)
- LOCATION-2: Access Control (CWE-284)
- LOCATION-3: Authentication Module (CWE-287)

STEP 2: Identification of Domain Concepts and Relationships

Based on the identified resources and locations, we identified the domain concepts and relations as shown in the metamodel for CWE-307 (Figure 7). Few additions such as OperationAttemptTimeStamp are implied by the descriptions.

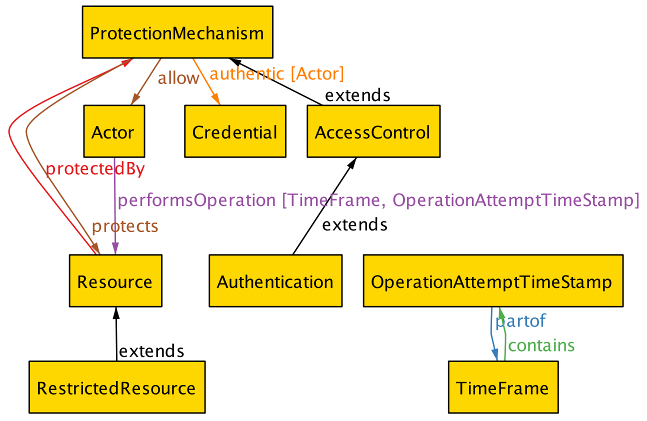


Figure 7: CWE-307 - Alloy metamodel

STEPS 3-4: Modeling and Checking Operational Behavior

The predicates representing the following operations are derived from the weaknesses:

- Accessing a resource at a time stamp in a time frame.

The un-desired behavior is specified by an assertion to check that multiple failed authentication time-stamped accesses do not occur within a given time frame (i.e., time points not in the same time frame). This assertion can be checked for a single actor (static) or over multiple states of an actor (dynamic) depending on how the specification is developed. Checking this assertion over multiple actor states produced counter-examples, which revealed interesting situations that the CWE description and its parents have left unspecified. For example, if there is a successful access among (or after) multiple failed attempts within the time frame, should that be allowed? Cases like these suggest possible improvements needed in the CWE description or possible new CWE descriptions that specify such cases

Executable Alloy specifications for this and other models in this paper can be downloaded from the following location: http://faculty.ist.unomaha.edu/rgandhi/st/alloy.

VIII. Conclusion and Discussions

Through case studies we make the following observations:

1. Many constraints in an Alloy specification for an abstract CWE can be extended or reused for CWEs that are its children. Such reuse enables a clearer definition distinguishing specialized weaknesses from more general weaknesses. This approach reduces the burden for Alloy specification developers and users through the use of patterns. In our prior work, patterns were used to capture constraints expressed in regulatory requirements [3].
2. Lightweight formalization allows to systematically aggregate all aspects of a weakness definition that are often implicitly considered in describing specific CWEs. This is particularly the case with weaknesses that are classified as *base* or *variant*, as they assume the context and understanding of parent *class* weaknesses.
3. Dynamic models in Alloy allow constraints specified over a sequence of states. Such models are more appropriate to capture traces that lead to weaknesses.

While others have investigated the use of formal specification to help detect software vulnerabilities (e.g., [1]), the goal of our approach is to facilitate a common understanding of software weaknesses, rather than being used to directly look for software weaknesses. The lightweight specifications can drive downstream weakness detection and removal activities using static or dynamic scanning tools.

The CWEs discussed here are chosen as part of a pilot project at NIST to promote a precise and accurate definition. Our focus is on some of the most egregious weakness, with enough diversity. Our ongoing and future work includes, associating fragments of Alloy specification with concepts in a semantic template [4], such that more complex situations can be modeled and analyzed in the context of a specific vulnerability. Such modeling can assist in understanding the appropriateness of a fix for a vulnerability or its interference with other system behaviors. It is our hypothesis that encoding CWE definitions using lightweight formalizations will improve their interpretation and integration in a given system context. Future case studies related to this hypothesis will evaluate the generalizability of our approach; provide opportunities to explore inconsistencies in CWE definitions; and improve ways to statically or dynamically detect weaknesses in software.

Acknowledgment

This research is partially funded by DoD/AFOSR, NSF FA9550-07-1-0499, "High Assurance Software" and NIST 70NANB12H013, "Developing Precise and Accurate Descriptions of Common Software Weaknesses".

We also acknowledge several constructive reviews from Dr. Paul E. Black, NIST, on early drafts of this paper.

References

[1] M. Almorsy, J. Grundy, A.S. Ibrahim, "Supporting Automated Vulnerability Analysis Using Formalized Vulnerability Signatures," Intl. Conf. on Automated Soft. Eng. (ASE 2012).

[2] B. Bruegge, A. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall, 2009.

[3] R.A. Gandhi, M. Rahmani, "Early Security Patterns: A Collection of Constraints to Describe Regulatory Security Requirements," RePa 2012: Intl. Workshop on Requirements Patterns, Intl. Conf. on Requirements Engineering (RE 2012).

[4] R.A. Gandhi, H. Siy, Y. Wu, "Studying Software Vulnerabilities," CrossTalk, The Journal of Defense Software Engineering, Sept/Oct issue 2010.

[5] D. Jackson, Software Abstractions: Logic, Language, and Analysis. MIT Press, 2012.

[6] MITRE, "Common Weakness Enumeration," http://cwe.mitre.org/ (Accessed on Jan. 10, 2013).

[7] Y. Wu, H. Siy, R.A. Gandhi, "Empirical Results on the Study of Software Vulnerabilities (NIER Track)." International Conference on Software Engineering (ICSE 2011), May, 2011.