# A Security Analysis Framework Powered by an Expert System

**Maher Mohamed Gamal** mahergamal@gmail.com
*Computer Science Arab Academy of*
*Science, Technology and Maritime Transport*
*Cairo, Egypt*

**Dr. Bahaa Hasan** bahaa.hasan@asc-egypt.org
*Chairman & CEO of Arab Security*
*Consultants (ASC)*
*Cairo, Egypt*

**Dr. Abdel Fatah Hegazy** ahegazy@aast.edu
*Computer Science*
*Arab Academy of Science, Technology*
*and Maritime Transport*
*Cairo, Egypt*

**Abstract**

Today's IT systems are facing a major challenge in confronting the fast rate of emerging security threats. Although many security tools are being employed within organizations in order to standup to these threats, the information revealed is very inferior in providing a rich understanding to the consequences of the discovered vulnerabilities. We believe expert systems can play an important role in capturing any security expertise from various sources in order to provide the informative deductions we are looking for from the supplied inputs. Throughout this research effort, we have built the Open Security Knowledge Engineered (OpenSKE) framework 1, which is a security analysis framework built around an expert system in order to reason over the security information collected from external sources. Our implementation has been published online in order to facilitate and encourage online collaboration to increase the practical research within the field of security analysis.

**Keywords:** Security Analysis, Expert System, Vulnerability Analysis, Security Framework, Attacks.

## 1. INTRODUCTION

Probably any organization today will probably need to benefit from the productivity that computers bring by to many applications within the organization's field. Unfortunately, with this productivity, comes a great risk of being prone to computer security attacks due to any existing vulnerable or misconfigured software. This has led organizations today to leverage various security tools in order to keep up with the continuous threats to their valuable assets and services. Various security tools such as port scanners, anti-viruses, intrusion detection systems and similar programs have all proved their usefulness by providing network administrators with the necessary information in order to identify their systems' defects.

Unfortunately, the information revealed by these security tools mostly provides a very inferior study to how these scattered pieces of information form together a bigger meaning along with it's consequences. This is why well-funded organizations would hire highly specialized professionals (aka. Red Team [2]) in order to lay out all of the collected data and analyze any possible attack intents. They usually end up with a graph of how the present vulnerabilities on the systems can lead to one or more potential attacks. Thus, there is a dire need to gain a deeper understanding from the security reports and information that are being extracted by the deployed sentinels in order to fully understand what is really happening behind the scenes. For example, even if a port scanner does reveal some open ports on a specific host, that doesn't designate a real problem since we may have public services listening on these ports. On the other hand, having these ports open on this specific machine with no need can lead to unknown potential attacks. So let us dig deeper into how attacks are performed.

A security attack can be performed by executing one or more exploits according to what it needs in order to be accomplished. An *exploit* is a program that leverages one or more vulnerabilities located in any of the installed software in order to cause an unintended behavior on the target system.

Previous efforts have been made in order to describe the attack concepts and one that really inspired us was Templeton and Levitt's [1] effort where they modeled the components that constitute an attack and how they relate to each other. This way of thinking breaks down the notion of an attack into it's constituents. In doing this, we can start studying the requirements of an attack's component and it's effect on it's surrounding environment.

This is illustrated in Figure 1 where we have an attack that can be achieved by leveraging two exploits, each having it's own capability requirements. A capability here can be an open port, a file permission, a vulnerability in a specific library or program ...etc. Therefore, when Exploit 1's three capabilities are met, it can be executed, which consequently makes Exploit 2's capabilities satisfied and thus, Exploit 2 can be executed leading to more capabilities available.

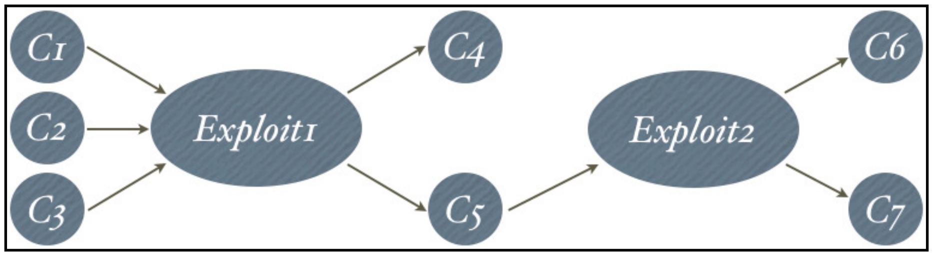

**FIGURE 1 :** Shows how an attack's components lead to each other through their capability requirements and offerings.

## 2. EARLY APPROACHES

Honestly, the field of security analysis isn't anew. A substantial amount of research has gone through several approaches to address this field. We will present the approaches that were relevant to our research in addition to what shortcomings that have been found in each of them.

### 2.1. Hard-Coding Vulnerability Checks

In 1987, Robert Baldwin published the first paper that proposed a rule based analysis method which was named Kuang [2]. Later came Daniel and Eugene to form this method into a practical security checker [3]. The efforts until then considered only vulnerabilities on a single host. Further

2 Red Team, http://en.wikipedia.org/wiki/Red_team

research was made to make Kuang work on multi-hosts on the same network, it was named NetKuang [4].

Unfortunately, the Kuang approach had the vulnerability checks hard-coded into it's implementation. Even though this approach was sufficient at it's time, nowadays, we are facing a rapid rate of vulnerability discoveries that render this approach impractical since any security checker nowadays needs to be able to import multiple formal specifications of vulnerabilities from various sources. In addition to this, we see that most of the attacks happening these days are a result from multi-staged sub-attacks on multi-hosts.
Nevertheless, we have borrowed the paradigm of using a rule-based method to analyze computer security in a similar fashion as we will see later on.

## 2.2. Model Checking

Model checking [5] is basically a state-transition system that is being checked whether it still satisfies a correctness condition. Applying model checking to network security can be in the form of modeling our systems as a state, where an attack on our systems would cause a transition from the current state to a different state. The state transition can be described in the form of the preconditions that need to be satisfied in order for the transition to be performed and the postconditions that would result from the transition. A full attack path would be a series of state transitions that would eventually violate the correctness condition (e.g. accessing classified data) upon being performed.

Unfortunately, as noted by Xinming [9], the drawback of model checking is that most state-transition sequences of the system are examined and with a large scale, this may eventually lead to a state-space explosion. In network security we only need to analyze what is feasible to be done from our current situation, not what could be done in the system's entirety disregarding it's achievability.

## 2.3. Attack Graph Analysis

The attack graph analysis approach, has previously attracted a hefty amount of research effort. The aim of this approach is to deliver an exploit-dependency graph which is identical to what we illustrated in Figure 1. The attack graph is used to analyze the possible actions the attacker can take in order to reach the target. Unfortunately, there has been several scalability problems as outlined in Lippmann's detailed review [6] of the previous publications on this topic. Although there has been several efforts listed in Lippmann's review that attempt to solve the scalability problems, we have decided not to take this approach as we have decided to leverage the power of a logical reasoner as we will see in the next section.

## 2.4. Logic-Programming

The logic-programming approach was introduced by Xinming [7] and Sudhakar [8] in their Datalog [3]-based security analysis framework MulVAL [9]. This approach has shifted our thinking of attack graphs into making them an outcome from the logical deductions performed over our domain understanding which is represented in the form of Datalog predicates. MulVAL produced full traces of the exploits that could be executed based on the experimented situations.

After looking into how MulVAL worked, we believe that MulVAL holds a couple of shortcomings which are listed below, though it still holds as one of the major inspirations to our research.

1. MulVAL is based on Datalog which can only provide an offline-mode of security analysis which means that in order for MulVAL to deduce any new information, it has to be asked for it. Although this is totally acceptable for what MulVAL was intended for (which is to generate attack traces), we believe this can be further improved to turn into an online

3 Datalog is a subset of Prolog, http://en.wikipedia.org/wiki/Datalog

analyzer where newly picked up security information is detected and fed into the analyzer which deduces new information.

2. MulVAL's domain modeling was in the form of Datalog predicates which on a large scale can turn out to be unmaintainable. A single entity's information is distributed among multiple predicates, which makes the understanding of the domain model harder to grasp and keep well maintained.
3. Datalog has mostly been used for academic purposes and we believe that in order for any open framework to be widely used and built upon, it has to be easily adoptable and the programming language used plays an important role in this.
4. In addition to the above, we intend to provide a publicly available open implementation of our framework that we hope would facilitate further research in this topic.

In the next section we will explore an Artificial Intelligence area called Expert Systems where we will see how it fits into the field of security analysis.

## 3. LEVERAGING AN EXPERT SYSTEM

Expert Systems [10] have long been a popular branch of Artificial Intelligence research. It's popularity has mainly stemmed from it's ability to reason over a problem based on it's current understanding of the situation.

To further understand what is meant by reasoning, it is when a system that holds some knowledge, is required to do or provide something that it was not explicitly informed with. Thus, the system must figure out what it needs to know from what it already knows.

### 3.1. The Structure of an Expert System

In order for expert systems to perform any kind of reasoning, they require the knowledge to be represented in a comprehensible format which would be known as it's knowledge representation. A collection of formalized pieces of information in a well-defined representation would be described as it's knowledge base and this forms the first of the two components that compose an expert system. The second part of an expert system is the logical reasoner which is the central brain that performs all of the necessary reasoning over the previously built knowledge base. The benefit of performing logical reasoning is that we can conclude new information, which can enlighten us and let us look at our situation with a better understanding.

### 3.2. Rule Chaining

```
IF
   <conditional expression(s)>
THEN
   <knowledge insert/update/retract statement(s)>
```

**FIGURE 2 :** An overly simplified structure of an expert system's rule syntax

The expert system's reasoner operates over well-defined domain rules. These rules can be thought of as *IF-THEN* statements as shown in Figure 2. Once the *IF* part of the statement is satisfied (i.e. the current situation implies that this rule should be fired ) the *THEN* part is computed which can introduce additional information that could be useful to us, plus it manipulates the knowledge base which can recursively cause more rules to be fired and thus, we end up with what we call, *forward chaining*.

An another way in which expert systems can operate, is called *backward chaining*. Here the expert system tries to prove whether a goal can be reached from the current understanding of the situation. This is mainly done be reversing the way the rules are traversed and this is what was adopted by the MulVAL [9] authors by using the Datalog language.

## 3.3. An Analogy between Templeton's Model and an Expert System

Comparing Templeton's [1] attack model to how an expert system's reasoning works, it is obvious how expert systems fit elegantly. As illustrated in Figure 3, Templeton's attack concept is represented in the expert system as a rule statement and the capabilities are represented as any piece of knowledge that is being required by any of the domain rules of the expert system.

| ***Requires/Provides Model*** | ***Expert System*** |
|---|---|
| Attack Concept | Rule Statement |
| Capability | Piece of Knowledge |

**FIGURE 3 :** Representation of Templeton's model in an Expert System

## 3.4. Choosing a Suitable Expert System

The real essence behind an expert system's logical reasoner is how it organizes the rules in an efficient manner to minimize the time taken to pass through all of the *IF* parts of the rules to evaluate them upon any updates to the knowledge base. Today's expert systems mainly build over the *Rete algorithm* [11] that was designed by Charles L. Forgy in 1982, which forms as one of the most efficient algorithms in maintaining and processing the rules of an expert system.

The expert system that we have found appropriate for our goal was Drools [4]. It's an open-source Rete-based expert system shell written in Java [5] which performs forward-chaining and features a very simple rule syntax that is easily comprehensible. We have favored Drools over others due to the following.

1. It supports forward-chaining which will highly aid in providing an online security analyzer that can receive a constant feed of security events.
2. The domain model is described as an object-oriented design which allows us to highly describe our domain problem with all possible relations.
3. It's rule syntax is very simple which will highly encourage security experts to contribute in writing the security rules.
4. Drools is built over Java which we believe is one of the most popular development platforms available today.
5. The Drools project is actively maintained and well documented.

The goal of this research effort is to leverage Drools as our expert system to capture any possible security knowledge, whether it's from an expert's technical expertise or security advisories in addition to the current network situation in order to conclude meanings that weren't perceptible before. On our way to achieve this, we will be facing the notion of formalizing the information that's being fed into Drools. After that, we will inspect how the Expert rules are written. Finally, we will conclude our work with the results that we have reached and what we envision to be possible for future development.

[1] 4 Drools, http:
5 Java, http://en.wikipedia.org/wiki/Java_(programming_language)

## 4. INCORPORATING OPEN COMMUNITY-DRIVEN STANDARDS

One of the greatest challenges in building our security analysis framework was coming up with a reasonable knowledge base to work on. Long ago, the learnings of computer security have been weakly formalized or even verified for it's correctness. Even with security advisories reporting the latest vulnerabilities, they were sent out as free text to mailing lists which are difficult to depend on in our research.

Today, with the rise of several community-driven efforts under the *Making Security Measurable Initiative* [12] to establish common standards in order to unify the understanding of several aspects of computer security, we decided to take it a chance to incorporate what is possible from their publicly available XML data-sets into our security analysis framework.

Below is a brief listing and description of what suited our framework's initial scope.

1. *Common Vulnerabilities and Exposures Enumeration (CVE)*

   The CVE standard is a constantly updated comprehensive dictionary of security vulnerabilities and exposures. These are specific to public releases of widely used software. We will be referring to CVE identifiers whenever we refer to specific vulnerabilities.

2. *Common Platform Enumeration (CPE)*

   The CPE standard proposes a unified naming convention for systems, platforms and software packages, in order to avoid any ambiguity when referring to a specific package version on a specific operating system.

3. *Open Vulnerability and Assessment Language (OVAL)*

   The OVAL scanner sweeps through the inspected systems searching for vulnerabilities that match any predefined signatures and reports them in the form of standard CVE identifiers or identifiers from the National Vulnerability Database (NVD) [6].

4. *Common Weakness Enumeration (CWE)*

   The CWE describes the software security weaknesses whether it's in architecture, design or code. Weaknesses can be thought of as the root causes of vulnerabilities. Each weakness is linked to it's observed vulnerabilities.

5. *Common Attack Pattern Enumeration and Classification (CAPEC)*

   The CAPEC provides a higher level view to the weaknesses in CWE and vulnerabilities in CVE/NVD. It shows the attack patterns that the attacker can perform by leveraging the weaknesses found in our systems in order to perform any unintended behavior.

## 5. INTRODUCING OPENSKE, THE FRAMEWORK

Our publicly available research-oriented framework, the Open Security Knowledge Engineered 7 (pronounced as open-skee) has been designed in order to leverage Drools as it's expert system in addition to surrounding it with all of the necessary auxiliaries to facilitate it's goal of analyzing network security. We have decided to open-source the implementation and provide it publicly in

6 National Vulnerability Database, http://nvd.nist.gov/
7 OpenSKE, http://code.google.com/p/openske

order to facilitate practical collaboration and to provide a basis for future research that can build over it. Before we delve into the framework's internals, let us list what we expect the framework to serve us.

1. Identify which of our assets may be affected by the present vulnerabilities in our systems.
2. Provide a list of CWE weaknesses (root causes) behind the existing CVE vulnerabilities.
3. Provide a list of CAPEC attack patterns that can be executed based on the existing weaknesses and vulnerabilities.
4. Report any activity performed by any attacker(s).

## 5.1. OpenSKE's Inputs and Outputs

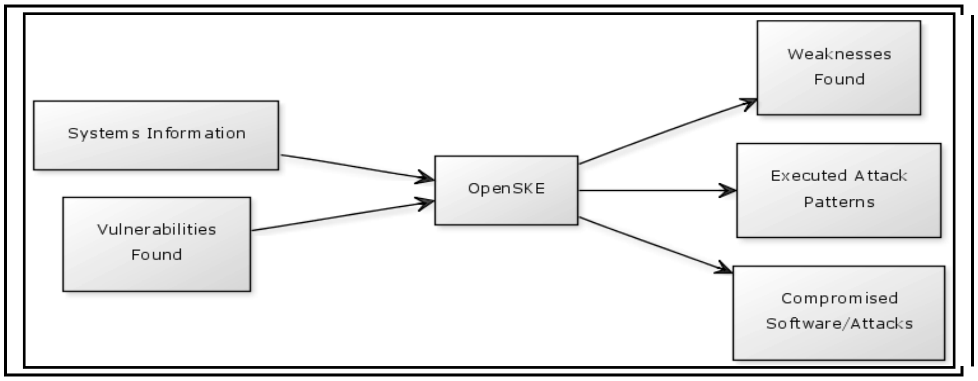

**FIGURE 4 :** An overview of OpenSKE's inputs and outputs

- ***OpenSKE's Inputs***

  1. *Systems Information*

     A full enumeration of all of the networked hosts with all of their user accounts, assets and applications running on them, ...etc.

  2. *Vulnerabilities Found*

     We will be supplying OpenSKE with the results of our vulnerability scanners such as the OVAL scanner to support pinpointing the vulnerabilities in our systems.

- ***OpenSKE's Outputs***

  1. *Weaknesses Found*

     A list of all weaknesses described by the CWE that have been satisfied by the current vulnerabilities in our systems.

  2. *Executed Attack Patterns*

     This constitutes a list of the CAPEC attack patterns that have been executed on our systems due to having their requirements satisfied.

  3. *Compromised Software/Assets*

     Software that has been attacked or assets that have been accessed or destroyed are reported.

We believe that by providing the above, we will be putting ourselves on a higher ground with the necessary information, tools and techniques to understand how secure our systems are.

## 6. OPENSKE'S DOMAIN MODEL

The domain model has been described in the form of Java classes that are inter-related together. We have tried to keep the domain model thorough enough to identify each participating entity along with it's obvious internals and behavior. Though, we haven't tied it to any particular vendor in order to keep it as independent as possible. In our illustration [8] of the domain model we will be showing a breakdown of the UML design to explore how the entities relate together. Throughout this chapter, we will see the notion of an entity's security state. This merely indicates the entity's condition (*unknown*, *safe*, *risky* or *compromised*) from a security point of view.

### 6.1. Hardware Domain Model

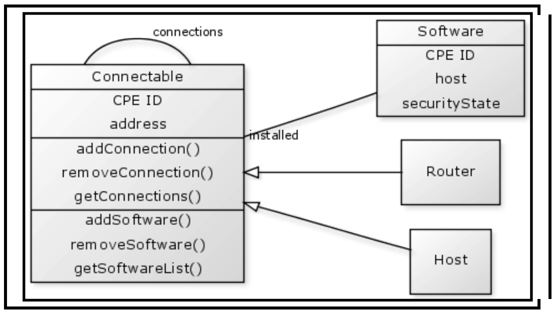

**FIGURE 5 :** Hardware Domain Model

Starting with the hardware domain model illustrated in Figure 5, we find that our hosts and routers inherit from a `Connectable` abstract class which provides the ability to interconnect hardware devices to each other and add software. Any possible `Connectable` subclass can contain software, the relation is also clearly shown in the diagram and has also been facilitated through the APIs. The `Host` and `Router` classes can expand, retract or override whatever functionality inherited from the `Connectable` class.

### 6.2. Assets Domain Model

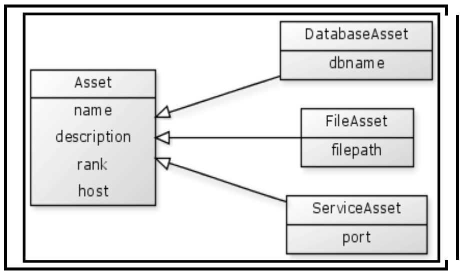

**FIGURE 6 :** Assets Domain Model

Next are assets, which are the most valuable resources maintained throughout the systems within an organization. Their types vary as illustrated in Figure 6 which shows different types of assets whether it's a database, file or service that needs to be secured throughout it's lifetime. The definition of an asset may be vague at times, but generally, it is anything that is important to the owning organization or person.

8 All UML diagrams presented here have been drawn using yUML ( http://yuml.me )

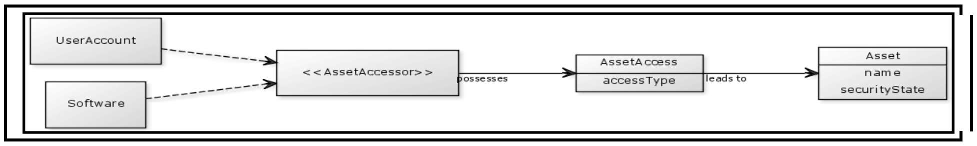

**FIGURE 7 :** Asset Accessibility

In addition to having our assets modeled, we had to approach how other entities would access the assets. Figure 7 shows how any entity (such as `Software` or `UserAccount`) that implements that `AssetAccessor` interface can easily possess `AssetAccess`es to any of the available assets with respect to the `AccessType` given.

## 6.3. Software Domain Model

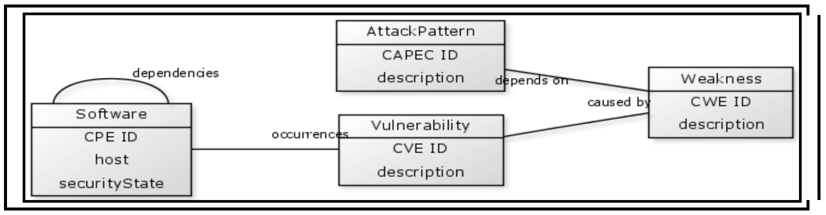

**FIGURE 8 :** Software & Vulnerabilities Domain Model

Software in OpenSKE is identified using their corresponding CPE identifiers to avoid any naming collisions and as illustrated in Figure 8, any software can contain occurrences of vulnerabilities identified by their CVE identifiers. Vulnerabilities are linked to their CWE weaknesses by looking up the CWE data-set. Software can depend on other software and thus, any piece of software that depends for example, on a faulty library, is potentially vulnerable as well. Implementation-wise, we have made the weaknesses of a software accessible from the software rather than having to traverse the software → vulnerabilities → weaknesses chain. This would help the reasoning to be more effective.

## 6.4. User Security Domain Model

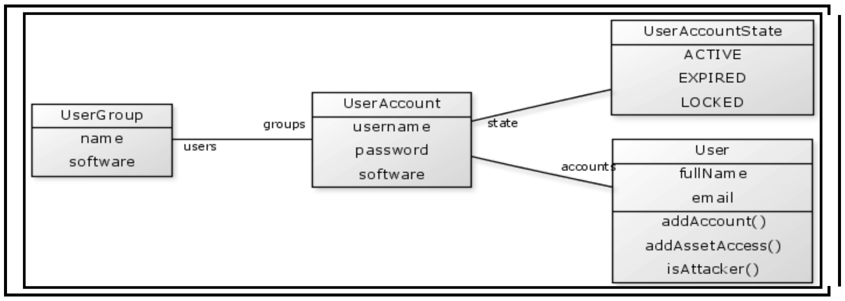

**FIGURE 9 :** User Security Domain Model

As shown in Figure 9, users in OpenSKE can possess multiple accounts on one or more hosts. These accounts may be further organized into groups as we usually see in most common applications. A user account is merely the credentials needed to gain an access level on a running software and it's state can be one of the listed values in Figure 9.

Now that we have covered our domain model, next we will see how we will leverage this domain model in our expert rules which will attempt to uncover more useful information from our initial understanding.

## 7. DEFINING THE SECURITY DOMAIN RULES

The most important possessions of an IT infrastructure are it's assets. Actually, without the assets, the systems won't be of much potential to be viable to attacks, unless the attacker intends to just gain more ground to launch further additional attacks and in this case, the hosts themselves can be considered as assets. Thus, we really need to understand where our assets stand from a security point of view. Before we delve into the rules of our system, let's quickly illustrate an example of what constitutes a Drools rule and how does the inference work over these rules.

### 7.1. An Anatomy of a Drools Rule

```
rule "Print assets that have become risky"
  when
    # Match an asset that has it's security state RISKY
    # Bind the matched Asset instance to the variable $asset
    $asset : Asset(
      securityState == SecurityState.RISKY
    )
  then
    print("[LOGGING] Asset '%s' has become risky !", $asset.getName());
end
```

**FIGURE 10 :** Drools Rule Example ( Printing risky assets )

Figure 10 shows a very simple example of a Drools rule which is executed if any asset has become in a risky state (this happens when it's surrounded by vulnerable software, we'll see this later). The rule consists of two parts, the left-hand side which is the *when* part and the right-hand side which is the *then* part. The left-hand side contains the patterns to be matched upon the facts in our knowledge base. If our pattern successfully matches some facts, it is bound to the variables we have specified ( in this case it's `$asset` ), then the right-hand side is executed with the bound data ( in this case we are printing the names of the risky assets using common Java syntax). We will illustrate a couple of rules from OpenSKE which should give a good understanding on how the rules and reasoning work.

## 7.2. Identifying Vulnerable Software

```
rule "Mark vulnerable software or those that depend on it as risky"
  when
    # Match any vulnerable software
    $sw : Software( vulnerabilities.size > 0 )
    or
    (
      # Match any existing risky software
      $dep : Software(
        securityState == SecurityState.RISKY
      )
      and
      # Match any software that depends on $dep
      $sw : Software(
        dependencies contains $dep
      )
    )
  then
    # Update the software as risky
    $sw.setSecurityState(SecurityState.RISKY);
end
```

**FIGURE 11 :** Identifying vulnerable software as risky

Figure 11 shows the rule that identifies risky software by checking if it has vulnerabilities or if it depends on any previously identified risky software. Upon having any possible facts that match the *when* part of the rule, the matched software **`$sw`** is updated to being risky from a security point of view, this is used later in the next section to deduce whether this risky software may affect our assets.

## 7.3. Identifying Assets that are Threatened

```
rule "Mark assets that are surrounded by risky software as risky"
  when
    # Match any asset available on our systems
    $asset : Asset()
    and
      exists(
        # Match any risky software on the same asset's host
        Software(
          host == $asset.host ,
          securityState == SecurityState.RISKY
        )
        or
        (
          # Match any neighbor host to the asset's host
          $neighbor : Host(
            connections contains $asset.host
          )
          and
          # Match any risky software on the neighbor host
          Software(
            host == $neighbor ,
            securityState == SecurityState.RISKY
          )
        )
      )
  then
    # Update the asset as risky
    $asset.setSecurityState(SecurityState.RISKY);
end
```

**FIGURE 12 :** Identifying assets surrounded by vulnerable software as risky

Figure 12 illustrates how we mark which of our assets are in jeopardy because of any surrounding vulnerable software on the same host or from a neighbor host. If we end up with a matched asset here, it is updated as being in a risky security state.

The next set of rules will be tackling a selection of some CAPEC attack patterns and how their preconditions and postconditions are modeled in OpenSKE.

## 7.4. Modeling the CAPEC Attack Patterns

```
rule  "CAPEC-1  :  Accessing  Functionality  Not  Properly  Constrained  by
ACLs"
  when
    # We have an attacker
    $attacker : User(
      attacker == true
    )
    # A software that the attacker may target
    $software : Software()
    # The software contains any of the listed weaknesses
    exists(
      Weakness(
        software == $software ,
        identifier in ("CWE-285", "CWE-732",
                       "CWE-276", "CWE-693",
                       "CWE-721", "CWE-434")
      )
    )
    # Attacker has an active user account on this software
    exists(
      UserAccount(
        software == $software ,
        state == UserAccountState.ACTIVE
      ) from $attacker.getAccounts()
    )
    # The attacker can still reach this software
    eval(
      $attacker.getHost().canReach(
        $software.getHost()
      )
    )
    then
      print("[CAPEC-1] Attacker '%s' can gain un-authorized
accessibility on software '%s'", $attacker.getFullName(),
                $software.toString());
end
```

**FIGURE 13 :** CAPEC-1, Accessing Functionality Not Properly Constrained by ACLs

CAPEC-1 [9] which is the first attack pattern in the CAPEC dictionary shows how an attacker can gain unauthorized access to functionality that should have been protected for higher authorized people. The rule states that if we have an attacker (which is a normal `User` in OpenSKE ) with an active `UserAccount` on a `Software` that contains one of the listed `Weakness`es and that the attacker can reach this software, then the attacker can possess unauthorized functionality on the target software. The CAPEC lists possible ways to mitigate the situation, but within the scope of security analysis, this may be useful when designing countermeasures to be taken against the attacks. CAPEC-1's post-conditions in specific, cannot be speculated in OpenSKE, since it highly depends on the nature of the software application being analyzed, which as you can see here, is totally unknown (i.e. software's features aren't modeled yet).

9 CAPEC-1, http://capec.mitre.org/data/definitions/1.html

```
rule "CAPEC-2 : Inducing Account Lockout"
  when
    # We have an attacker
    $attacker : User(
      attacker == true
    )
    # A software that the attacker may target
    $software : Software( accounts.size() > 0 )
    # The software contains any of the listed weaknesses
    exists(
      Weakness(
        software == $software ,
        identifier in ("CWE-400")
      )
    )
    # The software has any active user account
    $userAccount : UserAccount(
      software == $software ,
      state == UserAccountState.ACTIVE
    )
    # The attacker can reach this software
    eval(
      $attacker.getHost().canReach(
        $software.getHost()
      )
    )
    then
      # Lock the user account
      $userAccount.setState(UserAccountState.LOCKED);
      print("[CAPEC-2] Attacker '%s' has attacked the user account '%s'
on software '%s' and resulted in the account being locked",
$attacker.getFullName(), $userAccount.getUsername(),
$software.toString());
end
```

**FIGURE 14 :** CAPEC-2, Inducing Account Lockout

CAPEC-2 [10] involves the attacker targeting the supposedly defensive mechanism being employed in some authentication systems which is to lockout an account if it's login attempts have passed a number of tries. The rule mentions that if an attacker can reach a software with any active user accounts and that the software possesses the weakness described by CWE-400 [11], then the consequence is that the attacker can keep trying to perform multiple random logins until the account is locked (even though the original account owner had nothing to do with this).

10 CAPEC-2, http://capec.mitre.org/data/definitions/2.html
11 CWE-400, http://cwe.mitre.org/data/definitions/400.html

```
rule "CAPEC-7 : Blind SQL Injection"
  when
    # We have an attacker
    $attacker : User(
      attacker == true
    )
    # A software that the attacker may target
    $software : Software()
    # The software contains any of the listed weaknesses
    exists(
      Weakness(
        software == $software ,
        identifier in ("CWE-89", "CWE-209",
                       "CWE-74", "CWE-20",
                       "CWE-390", "CWE-697",
                       "CWE-713", "CWE-707")
      )
    )
    # The attacker can reach this software
    eval(
      $attacker.getHost().canReach(
        $software.getHost()
      )
    )
    then
      # Attacker gains access to the database assets from this software
      # We have assigned a random asset with a random access type for
      # the simulation
      $attacker.addAssetAccess(
        new AssetAccess(
          $software.getRandomAsset(AssetType.DATABASE),
          $attacker,
          AssetAccessType.getRandomValue()
        )
      );
      print("[CAPEC-7] Attacker '%s' has gained '%s' access to database
'%s' through SQL injection on software '%s'",
        $attacker.getFullName(),
        $attacker.getRecentAssetAccess().getType(),
        $attacker.getRecentAssetAccess().getAsset().getName(),
        $software.toString()
      );
end
```

**FIGURE 15 :** CAPEC-7 : Blind SQL Injection

The CAPEC-7 [12] is one of the common attack patterns that we see applicable to online web applications. It is similar to the previously illustrated attack patterns, but the difference lies in the consequences of the attack, which in this case, grants the attacker an `AssetAccess` to one of the `DatabaseAsset`s that the matched software possesses. The randomization performed in the value selection of the rules has been done in order to avoid fixating scenarios. In reality, this is solely up to the attacker proficiency to get the best benefits out of the attack pattern being executed.

12 CAPEC-7, http://capec.mitre.org/data/definitions/7.html

```
rule "CAPEC-16 : Dictionary-based Password Attack"
  when
    # We have an attacker
    $attacker : User(
      attacker == true
    )
    # We have an installed and running software with user accounts
    $software : Software( accounts.size > 0 )
    # The software contains any of the listed weaknesses
    exists(
      Weakness(
        software == $software ,
        identifier in ("CWE-521", "CWE-262",
                       "CWE-263", "CWE-693")
      )
    )
    # The attacker doesn't have an account on this software
    not(
      exists(
        UserAccount(
          software == $software ,
          state == UserAccountState.ACTIVE
        ) from $attacker.accounts
      )
    )
    # The attacker can reach this software
    eval(
      $attacker.getHost().canReach(
        $software.getHost()
      )
    )
    then
      # The attacker gained a user account
      $attacker.addAccount(
        $software.getRandomAccount()
      );
      print("[CAPEC-16] Attacker '%s' has hacked account '%s' on
software '%s'",
        $attacker.getFullName(),
        $attacker.getRecentAccount(),
        $software.toString()
      );
end
```

**FIGURE 16 :** CAPEC-16 : Dictionary-based Password Attack

CAPEC-16 [13] is one of the most commonly used attack patterns on the Internet since most people use normal words for their passwords. The attacker here is attacking a software that possesses at least one active user account and suffers from one of the listed weaknesses. Upon executing the attack, the attacker is granted an account on the targeted software. Now that we have illustrated how OpenSKE's rules work. We will be seeing their application in an experiment in the next section.

13 CAPEC-16, http://capec.mitre.org/data/definitions/16.html

## 8. EXPERIMENTING OPENSKE

### 8.1. The Experiment Setup

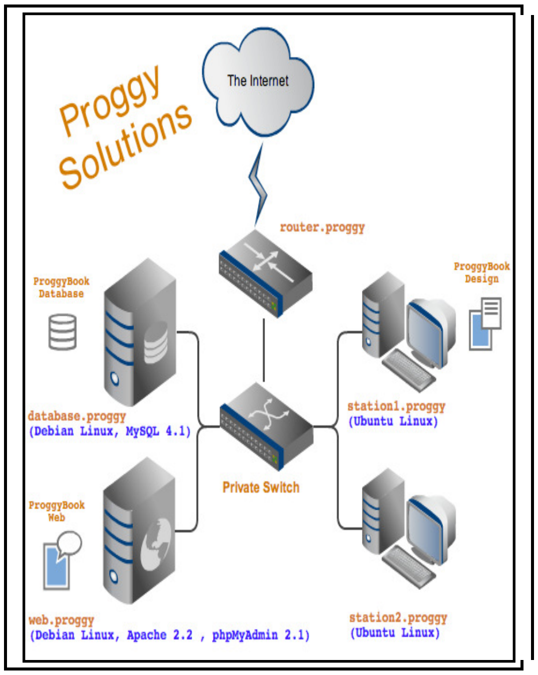

**FIGURE 17 :** Proggy Solutions Infrastructure [14]

Proggy Solutions is a novel startup company specialized in providing public web-services. Their most popular service, ProggyBook which is a public social-networking website running on their self-hosted Apache [15] web server and MySQL [16] database server as outlined in Figure 17. In addition to these supporting services, they use phpMyAdmin [17] as a web interface to manage their MySQL databases. Usually, there is an on-call off-site support personnel which troubleshoots any occasional issues using several tools including phpMyAdmin.

| ***Name*** | ***Type*** | ***Location*** |
|---|---|---|
| ProggyBook Web | Service Asset | web.proggy |
| ProggyBook DB | Database Asset | database.proggy |
| ProggyBook Design | File Asset | station1.proggy |

**FIGURE 18 :** Proggy's Valuable Assets

Figure 18 outlines Proggy's valuable assets and we will be taking these assets as our targets in this experiment. We will also assume that Proggy's systems contain the weaknesses listed below in Figure 19.

14 The infrastructure diagram was created using Gliffy ( http://www.gliffy.com )
15 Apache Web Server, http://httpd.apache.org/
16 MySQL Database Server, http://www.mysql.com/
17 phpMyAdmin, http://www.phpmyadmin.net/

| ***Software*** | ***CWE ID*** | ***Description*** |
|---|---|---|
| ProggyBook Web 1.0 | CWE-20 [18] | Improper Input Validation |
| ProggyBook Web 1.0 | CWE-285 [19] | Improper Access Control |
| ProggyBook Web 1.0 | CWE-400 [20] | Uncontrolled Resource Consumption |
| phpMyAdmin 2.1 | CWE-521 [21] | Weak Password Requirements |

**FIGURE 19 :** Proggy's Systems' Weaknesses

The experiment was run on a casual Apple MacBook Pro with the following specifications.

| | |
|---|---|
| ***Processor*** | 2.53 GHz |
| ***Memory*** | 4 GiB |
| ***Java Virtual Machine*** | Java(TM) SE Runtime Environment 1.6.0_20 |
| ***Drools Expert System*** | 5.1.1 |

**FIGURE 20 :** Platform Specifications

### 8.2. Output Results

The execution of OpenSKE resulted in uncovering the consequences of the assumed weaknesses mentioned in section 8.1. Figure 21 shows the detailed output of the experiment execution.

```
~/Workspaces/OpenSKE/openske > ./openske

>> Building OpenSKE...

>> Running OpenSKE's console...

Welcome to OpenSKE (JVM: 1.6.0_20) !
Type 'help' for help
openske> start

[OPENSKE] Running OpenSKE engine...

[OPENSKE] Initializing Drools Knowledge Base...

[OPENSKE] Loading 4 rule files...
- Loading './openske-expertise/src/main/resources/com/openske/rules/Assets.drl'
- Loading './openske-expertise/src/main/resources/com/openske/rules/Attack Patterns.drl'
- Loading './openske-expertise/src/main/resources/com/openske/rules/Logging.drl'
- Loading './openske-expertise/src/main/resources/com/openske/rules/Software.drl'

[OPENSKE] Adding 1 compiled knowledge packages to the knowledgebase...
         - Knowledge Package com.openske.rules (9 rules)

[OPENSKE] Inserting the facts into the knowledge base...

[DROOLS] Activation Created : Detect the reachability of an attacker ( if any )
[DROOLS] Activation Created : Detect the reachability of an attacker ( if any )
[DROOLS] Activation Created : Detect the reachability of an attacker ( if any )
[DROOLS] Activation Created : CAPEC-2 : Inducing Account Lockout
```

18 CWE-20, http://cwe.mitre.org/data/definitions/20.html
19 CWE-285, http://cwe.mitre.org/data/definitions/285.html
20 CWE-400, http://cwe.mitre.org/data/definitions/400.html
21 CWE-521, http://cwe.mitre.org/data/definitions/521.html

```
[DROOLS] Activation Created : CAPEC-7 : Blind SQL Injection
[DROOLS] Activation Created : Detect the reachability of an attacker ( if any )
[DROOLS] Activation Created : CAPEC-16 : Dictionary-based Password Attack
[DROOLS] Activation Created : Detect the presence of an attacker

[OPENSKE] Firing all rules...

[DROOLS] Activation Fired : Detect the presence of an attacker
[LOGGING] Attacker 'Mr. X' detected on host 'attacker.proggy'

[DROOLS] Activation Fired : CAPEC-16 : Dictionary-based Password Attack
[CAPEC-16] Attacker 'Mr. X' has breached account 'admin' on software
'cpe:/a:phpmyadmin:phpmyadmin:2.1' through a dictionary-based password attack

[DROOLS] Activation Fired : Detect the reachability of an attacker ( if any )
[LOGGING] Attacker 'Mr. X' can reach software 'cpe:/a:proggysolutions:proggyweb:1.0'

[DROOLS] Activation Fired : CAPEC-7 : Blind SQL Injection
[CAPEC-7] Attacker 'Mr. X' has gained 'READ_WRITE' access to database 'ProggyBook
Database' through SQL injection on software 'cpe:/a:proggysolutions:proggyweb:1.0'

[DROOLS] Activation Fired : CAPEC-2 : Inducing Account Lockout
[CAPEC-2] Attacker 'Mr. X' has attacked the user account 'admin' on software
'cpe:/a:proggysolutions:proggyweb:1.0' and resulted in the account being locked

[DROOLS] Activation Fired : Detect the reachability of an attacker ( if any )
[LOGGING] Attacker 'Mr. X' can reach software 'cpe:/a:apache:apache:2.2'

[DROOLS] Activation Fired : Detect the reachability of an attacker ( if any )
[LOGGING] Attacker 'Mr. X' can reach software 'cpe:/a:phpmyadmin:phpmyadmin:2.1'

[OPENSKE] Engine took 4.21 seconds !
```

**FIGURE 21 :** OpenSKE's experiment output.

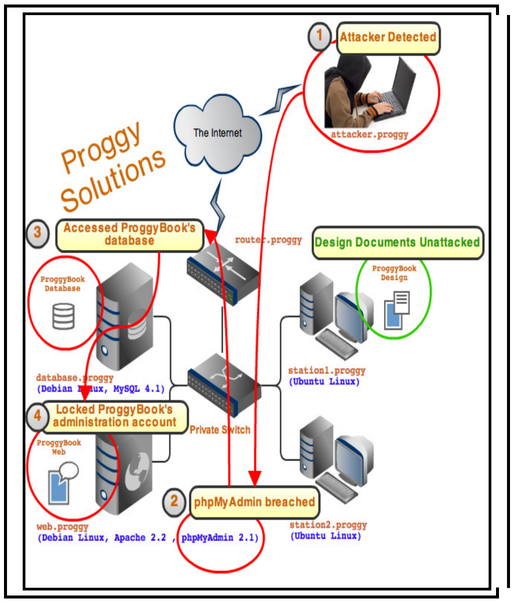

**FIGURE 22 :** OpenSKE's experiment output manual visualization

We have highlighted the important sections of the output which indicate the deductions made by OpenSKE in addition to mapping them on the infrastructure diagram in Figure 22. We will see that the first rule fired was the detection of the attacker *Mr. X* at `attacker.proggy`. The attacker attempted a dictionary-attack on the public phpMyAdmin interface and since phpMyAdmin held user accounts with default passwords ( such as root/root or admin/admin ) it was very easy for the attacker to gain access to the user accounts available. The attacker then attempted a SQL injection attack on the ProggyBook website which resulted in a READ_WRITE [22] access to the ProggyBook database. In addition to this, the ProggyBook developers thought it was a good idea to lock accounts on the third login failure. The attacker took advantage of this and attacked the admin account which resulted in the administration section being locked.

The total time taken for the initial execution was **4.21** seconds in performing the following items.

1. Initializing the knowledge base
2. Loading the rules
3. Inserting the facts
4. Firing the rules and updating the knowledge base accordingly

OpenSKE took way lesser time in consequent runs ( by doing a `restart` from the OpenSKE console ), due to the optimizations that are constantly performed by the Java Virtual Machine throughout the execution.

## 9. COMPARATIVE EVALUATION

In order to see how valuable the deductions of OpenSKE are, let us consider how do the results shown in the previous section compare to the findings of previous efforts.

### 9.1. The Execution Model

- The MulVal framework [9] operates in the form of a question-based approach. For example, after it's initialization, we start asking whether the attacker can execute code on a private server. When the backward-chaining process has been done, it replies with either the attack trace or none.
- The Model-Checking approach [5] operates after being provided with a system correctness condition that the model checker must stop upon having this condition no longer satisfiable. When the condition becomes violated, a counter-example is returned with the state transitions sequence that led to the system correctness violation.
- The OpenSKE framework operates in the form of an online approach which upon any update in the knowledge base, fires the rules waiting for such update which consequently performs additional deductions which is fed into the knowledge base again. Execution ends when no more deductions can be reached.

### 9.2. Syntactic Clarity

- The MulVal framework [9] is written in Datalog, thus any form of knowledge is represented as Datalog tuples similar to what we know from first-order logic. Object relations between the domain entities are defined in the predicate names, thus we can write as much as relations as possible as long as we can distinguish them clearly. Writing the rules requires reusing previous predicates in a recursive manner to achieve the desired understanding.
- The Model-Checking approach [5] was exercised using the Symbolic Model Verifier (SMV) [23] which had the system represented in arrays of booleans and literals. The initialization of the model had to be done with care in specifying the initial and next states

22 This was a randomly chosen access type for the SQL injection as it's totally based on the attempted SQL which isn't currently modeled in OpenSKE.
23 Symbolic Model Verifier, http://www.cs.cmu.edu/~modelcheck/smv.html

of each attribute. The original authors sacrificed the resulting program's clarity due to the limited feature set of the SMV checker.

- The OpenSKE framework leverages the Java programming language in order to represent the domain model in an object-oriented paradigm which allows modeling the domain in a highly precise and accurate manner. The rules are described in Drools's simple readable rule syntax which helps in comprehending the rule logic easily.

### 9.3. Scalability with Larger Infrastructures

- The MulVal framework [9] was benchmarked by it's authors and the results were impressive since MulVal was achieving the analysis in less than 1 second up until 400 hosts which then rises to 3.85 seconds with 1000 hosts.
- The Model-Checking approach [5] wasn't tested with a larger infrastructure than the one attempted by the original authors, but it was noted that the model checking approach may face a state-space explosion with a large number of state possibilities.
- The OpenSKE framework has been benchmarked with several infrastructure sizes and the results are shown in Figure 23.

| Infrastructure | Rules Load Time | Facts Load Time | Deductions Time |
|---|---|---|---|
| Proggy Solutions | 0.48 | 0.09 | 0.01 |
| 200 Hosts | 0.44 | 0.76 | 0.01 |
| 500 Hosts | 0.47 | 3.08 | 0.01 |
| 1000 Hosts | 0.42 | 15.06 | 0.03 |

**FIGURE 23 :** OpenSKE Benchmark Results (Time unit is seconds)

The timings shown in Figure 23 have been collected by running OpenSKE in `benchmark` mode for each infrastructure size. In `benchmark` mode, only high-level statistical output is written to the console to minimize any irrelevant any I/O. The reason why the deduction times are very negligible is because as soon as the facts are inserted, the rule matching is performed and the supposed rules to run are registered as to be activated once the rule firing is signaled.

## 10. OPENSKE'S CURRENT SHORTCOMINGS

Unfortunately, with the bright side shown in OpenSKE, it still suffers from a set of shortcomings which are listed below.

1. The domain model can be heavily expanded to further devices, systems and relations. The more we try to diversify, the more it will become applicable to more use cases.
2. We haven't mentioned how *time* plays an important role in the execution of attacks. Time has always been a crucial factor in many security related events. Temporal reasoning can facilitate this and OpenSKE can be further expanded to leverage the Drools Fusion [24] component in order to accomplish this.
3. We have supplied OpenSKE with Proggy's systems information in a manual manner, this can be automated by detecting the network topology and the running systems on a frequent basis.
4. Rules were being fired in an haphazard sequence which may at times make it incomprehensible, thus guiding them to go through a workflow process will highly organize the steps of OpenSKE's execution. OpenSKE can leverage Drools Flow [25] in order to achieve this.

24 Drools Fusion, http://www.jboss.org/drools/drools-fusion.html
25 Drools Flow, http://www.jboss.org/drools/drools-flow.html

5. We haven't considered retracting facts yet, but in reality this is unavoidable. For example, shutting down hosts or running services will dramatically change whatever has been deduced before, or at least prevent possible deductions to be made.

## 11. CONCLUSION

We envision that OpenSKE can be integrated with various security sentinels such as intrusion detection tools, firewalls, in which OpenSKE acts as the brain sitting at the back making sense of what's happening in order to take possible actions or provide the activities in a comprehensible manner to the administrators. Although our current ruleset has mostly described the CAPEC attack patterns, we believe further wisdom can be captured from security experts and formalized in order to deduce more in-depth meanings.

Finally, we highly welcome fellow researchers in the security analysis field to leverage the publicity [26] of OpenSKE's implementation in researching different topics and building further tools.

## 12. ACKNOWLEDGMENTS

Finally, I would like to thank my family for supporting me with all they had such that I would accomplish my research. My mentors have constantly provided me with wise and concise advice on various matters. Finally, I would like to thank the reviewers for their patience and effort in providing insightful input.

## 13. REFERENCES

5. Steven J. Templeton, Karl Levitt. “A Requires/Provides Model for Computer Attacks”. ACM Press, 2000.

6. Robert W. Baldwin. “Rule based Analysis of Computer Security”. MIT, 1987.

7. Daniel Farmer, Eugene H. Spafford. “The COPS Security Checker System”. Purdue, 1994.

8. Dan Zerkle, Karl Levitt. NetKuang – “A Multi-Host Configuration Vulnerability Checker”, California, 1996.

9. Ronald W. Ritchey, Paul Ammann. “Using Model Checking to Analyze Network Vulnerabilities”. IEEE Symposium on Security and Privacy, 2000.

10. R. P. Lippmann, K. W. Ingols. “An Annotated Review of Past Papers on Attack Graphs”. MIT 2005.

11. Xinming Ou. “A logic-programming approach to network security analysis”. Princeton University, 2005.

12. Sudhakar Govindavajhala. “A Formal Approach to Practical Network Security Management”. Princeton University, 2006.

13. Xinming Ou, Sudhakar Govindavajhala, Andrew W. Appel. “MulVAL: A Logic-based Network Security Analyzer”. Proceedings of the 14th USENIX Security Symposium, 2005.

14. Edward A.Feigenbaum. “Expert Systems : Principles and Practice”, The Encyclopedia of Computer Science and Engineering, 1992.

26 OpenSKE, http://code.google.com/p/openske

15. CL Forgy, Rete: “A fast algorithm for the many pattern/many object pattern match problem”. Artificial Intelligence, 1982.

16. Robert A. Martin. “Making Security Measurable and Manageable”, MILCOM 2008.

17. T. Tidwell, R. Larson, K. Fitch and J. Hale. “Modeling Internet Attacks”, IEEE 2001.

18. Sean Barnum, Amit Sethi. “Attack Patterns as a Knowledge Resource for Building Secure Software”, OMG Software Assurance Workshop: Cigital, 2007.