Using Abuse Case Models for Security Requirements Analysis

John McDermott and Chris Fox

Department of Computer Science

James Madison University

Harrisonburg, Virginia 222807

E-mail: {mcdermot, fox}@cs.jmu.edu

Abstract

The relationships between the work products of a security engineering process can be hard to understand, even for persons with a strong technical background but little knowledge of security engineering. Market forces are driving software practitioners who are not security specialists to develop software that requires security features. When these practitioners develop software solutions without appropriate security-specific processes and models, they sometimes fail to produce effective solutions.

We have adapted a proven object-oriented modeling technique, use cases, to capture and analyze security requirements in a simple way. We call the adaptation an abuse case model. *Its relationship to other security engineering work products is relatively simple, from a user perspective.*

1. Introduction

A valid security engineering process, as typified by the Common Criteria [1], is a complex activity involving many special work products: security objectives, security requirements, security policies, functional specifications, and security policy models. These work products are essential in a process that aims to create trustworthy information security products. But the work products and relationships between them can be hard to understand, even for persons with a strong technical background, but little knowledge of security engineering.

Security specialists use mathematical security models [6, 8] to understand security problems and find solutions for them. Use of these models is essential to the creation of trustworthy information security products. But these models are complex and subtle. They are not easily understood by persons who are not security specialists. They must be interpreted for the system to which they are applied. Security specialists can construct these interpretations, but the construction can be time consuming.

On the other hand, market forces are driving software practitioners who are not security specialists to develop software that requires security features. When these practitioners develop software solutions without appropriate security-specific processes and models [2], they sometimes fail to produce effective solutions [7].

While we do not have a solution to this problem, we have adapted a proven object-oriented modeling technique, *use cases*, to capture and analyze security requirements in a simple way. We call the adaptation an *abuse case* model. As we employ it, an abuse case model is considerably easier to understand than a mathematical security model. Its relationship to other security engineering work products is relatively simple, from a user perspective

2. Use Cases

Use cases are abstract episodes of interaction between a system and its environment. A use case characterizes a way of using a system, or a dialog that a system and its environment may share as they interact.

We define a *use case* as a specification of a type of complete interaction between a system and one or more *actors* (discussed below). A use case must be complete in the sense that it forms an entire and coherent transaction. For example, making a cash withdrawal at an ATM machine, placing a call on the telephone, or deleting a file from a file system, are examples of complete interactions with various sorts of systems that would qualify as use cases.

An abstraction of an external agent that interacts with the system is called an *actor*. Actors represent entities outside a system that interact with it in some way. Actors may be human or non-human. Actors may interact with a system by exchanging data with it, invoking one of the system's operations, or having one of the actor's operations invoked by the system. Actors are abstractions of actual individual users or systems typifying the roles played in system interactions. Some examples of actors are *Dispatcher*, *Clerk*, *Printer*, *Communications Channel*, and *Inventory System*.

A *scenario* is a description of a specific interaction between particular individuals. A use case abstracts scenarios that are instances of the same kind of interaction between a system and its actors. The following description is a scenario:

> Mary Smith places her bank card into an active ATM machine. The system prompts her for her PIN number, and she types 2384. The machine displays a transaction menu. Mary chooses a balance inquiry on her checking account. The system reports that she has $1329.67 in her account, and again displays the transaction menu. This time Mary chooses to end the interaction, and the system releases her card. Mary removes it and the system returns to its ready state.

A use case abstracts scenarios such as this to provide a general specification for similar interactions. The following illustrates an ATM balance inquiry transaction use case:

> A balance inquiry begins when a Customer inserts his or her bank card into an active ATM machine in its ready state. The system prompts for the Customer's PIN. The Customer types the PIN. If the PIN is incorrect, the system displays an error message and asks the Customer to try again. The Customer gets three tries. After the third failure, the system keeps the card, displays a message telling the Customer to see a teller, and after twenty seconds, returns to its active state. If the Customer enters a valid PIN, the system presents a transaction menu. The Customer chooses a balance inquiry on either checking or savings. The system displays the current balance and re-displays the transaction menu. This continues until the Customer chooses to terminate the interaction. The system releases the bank card. The Customer removes the card and the system returns to its ready state. If the bank card is not removed within 40 seconds, the system retrieves the bank card.

Notice that the use case describes the possible courses of events that may occur in various scenarios.

Use case models are documented using two notations: use case diagrams and use case descriptions. Use case diagrams are part of the Universal Modeling Language (UML), an industry standard collection of notations for analysis and design [5].

A use case diagram is a schematic representation of actors and a system's use cases. Each actor is represented by a stick figure; the actor's name appears near the figure. Use cases are represented by ovals with the name of the use case either below or within the oval. Finally, an association line connects the actors with the use cases in which they participate. These symbols are shown below, in Figure 1.

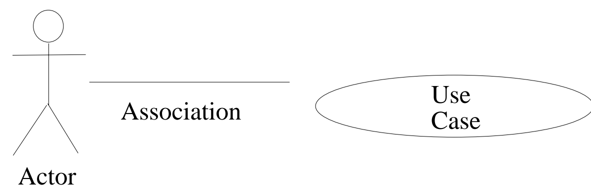

Figure 1. Use Case Diagram Symbols

Each use case should have an accompanying *use case description* that explains how the actors and the system interact. There is no standard notation or format for use case descriptions. Virtually any notation able to describe interactions between two or more entities may be used. Typically, use case descriptions are written in natural language. The simplest or most abstract use case description may be only a few sentences. More detailed use case descriptions are refined by adding details about the interaction and references to requirements for requirements traceability.

A special case of abstraction involves varying levels of detail about the interaction protocols between actors and the system. For example, an ATM balance inquiry use case description may simply state that the user identifies herself to the system, specifies a balance

inquiry transaction on one of her accounts, is informed of the balance in response, and then the user terminates the interaction. This description abstracts protocol details. A use case description with more protocol detail might go into the specifics of ATM cards, PIN numbers, the ATM display and menus, and the button presses the user makes to accomplish the transaction.

Use cases that abstract the details of interaction protocols are called *essential use cases*; those that include protocol details are called *real* or *implementation use cases* [3].

3. A Use Case Model For An Internet-Based Information Security Lab

Here is a complete use case model that will show how actors, use cases, associations, diagrams, and use case descriptions fit together. This model is a simplified version of the one we are using to design and set up an Internet-based information security lab in our own department.

An Internet-based information security lab, or *lab*, is a collection of systems and software used for teaching information security. Laboratory exercises give students practical experience with security vulnerabilities, security testing, and defenses. The students are not physically in the laboratory, but access it through the Internet. The lab comprises four kinds of entities: *servers*, *sources*, *targets*, and *exercises*. The first three are specially configured host systems in the lab. Servers provide presence for the students in the lab; servers do not participate in the exercises. Sources and targets participate in the exercises, with at least one source and target for each exercise. The exercises are either exploits or defenses, from the student's point of view. Each exercise has two parts: *documentation* and *implementation*. The documentation is provided by the instructor and usually consists of files and code samples that explain the exercise. Students are allowed to access the documentation for an exercise and are expected to construct and demonstrate an implementation. The instructor also provides a model solution which is not given to the students until the exercise is completed. Before each exercise, the lab is configured by an administrator. After the exercise is complete, the administrator restores the lab to an appropriate configuration.

3.1 Use Case Diagram

The use case diagram for the lab is show below as Figure 2. There are three actors and eight use cases.

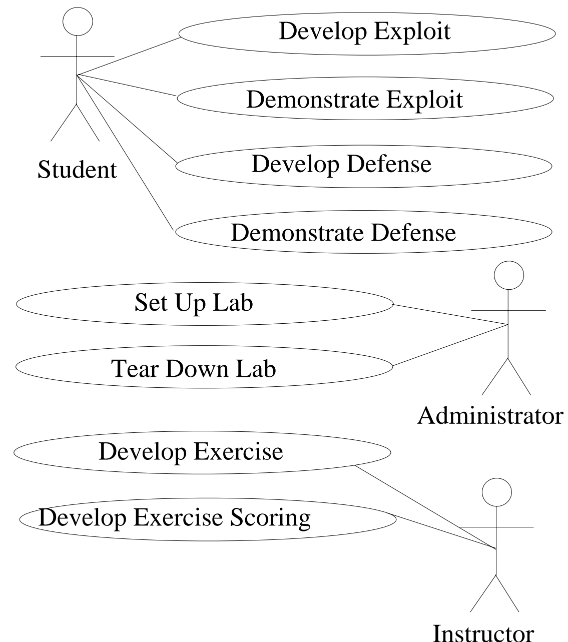

Figure 2. Use Case Diagram for an Internet-Based Information Security Laboratory

3.2 Use Case Descriptions

Here is a use case description from the model depicted in Figure 2. The first use case describes the interaction that takes place when a Student develops a security exploit as part of an assigned exercise. The second use case describes how a lab Administrator configures the hosts and networks of the lab for a particular exercise. Lab setup occurs for each exercise, because the exercise security configuration is quite frequently different from one that would be recommended for an operational system. Our descriptions are abstract and informal, as we would use for requirements elicitation. The actual lab will include several kinds of networks (Fast Ethernet, ATM, IPv4, IPv6) and hosts (Windows NT and Linux), and it is difficult to be specific in a small amount of space.

Develop Exploit

A Student first logs in to the lab server using a either a secure browser or command shell, from a remote location. The lab server authenticates the Student using a public key authentication algorithm and opens a session for the Student in her private workspace. The Student reads the description of the

exercise, due dates, assigned hosts, list of references, and scoring rules from a text file provided by the lab server at her request. The Student studies the references that are outside the lab. The Student requests references that are stored in the lab, from the lab server. The lab server returns the references that are permitted by the current security policy of the lab server. The Student then logs on to the target host, via the lab server. The Student requests pertinent configuration information from the target host. The target host returns the configuration information permitted by its local security policy and the security policy of the lab server. The Student studies the configuration of the target host. The Student may request an editor from the lab server. The lab server will provide one if the Student is authorized according to the security policy. The Student then uses the editor to write a plan for the exploit, copies of configuration files, and programs needed to demonstrate the exploit. Alternatively, the Student may use editors and software tools on her local system outside the lab and then request to upload them onto the lab server via the secure shell. If this is permitted by the current security policy, then the lab server accepts the files and stores them in the Student's workspace. When the Student is satisfied that her exploit is ready she requests that the lab server install the necessary files on the source machine. If this is permitted by the security policy then lab server installs the necessary files on the source machine. The Student then tests her solution against the target and modifies the configuration files, procedures, and programs until the exploit succeeds or the Student gives up. The Student then saves her solution files on the server and logs out.

4. Abuse Cases

We define an *abuse case* as a specification of a type of complete interaction between a system and one or more actors, where the results of the interaction are harmful to the system, one of the actors, or one of the stakeholders in the system. We cannot define completeness just in terms of coherent transactions between actors and the system. Instead, we must define abuse in terms of interactions that result in actual harm. A complete abuse case defines an interaction between an actor and the system that results in harm to a resource associated with one of the actors, one of the stakeholders, or the system itself. For example, it may be possible to define an interaction that reveals a session key to an actor that should not see the session key. However, we would not call this interaction an abuse case, because no actor has used the compromised key to divulge the contents of a message or make an unauthorized change to stored data. If we extend the interaction to include the posting of the key on a public web site then we have an abuse case.

A further distinction we make is that an abuse case should describe the abuse of privilege used to complete the abuse case. Clearly, any abuse can be accomplished by gaining total control of the target machine through modification of the system software or firmware. In many cases it is not necessary to abuse this much privilege in the real system, so we need to include abuse of privilege that is less than maximal. To guard against simple abuses, an abuse case should describe interactions involving the minimal abuse of privilege necessary to accomplish the harm intended by the abusing actor. However, in the real system, an attacker may employ more than minimal effort. For this reason, we describe the range of privileges that might be used to accomplish the abuse, up to the maximum we intend to deal with.

Finally, we also include a short description of the specific harm that will occur as a result of the abuse. This description should be in terms from the user's domain.

We can describe abuse cases using the same strategy as for use cases: use case diagrams and use case descriptions. We do not use any special symbols for abuse cases in diagrams, that is, an abuse case diagram is drawn with the same symbols as a use case diagram. This allows us to create abuse case specifications in standard notation such as UML and to use design tools such as *Rational Rose* [4]. We distinguish the two by keeping them separate and labeling the diagrams. Abuse cases are not shown on a use case diagram and use cases are not shown on an abuse case diagram. Abuse cases can also range in levels of abstraction and we use both essential abuse cases and real abuse cases.

The actors in an abuse case model are the same kinds of external agents that participate in use cases. However, they should not be the same actors. If a human that is represented by an actor from a use case might also act maliciously in the corresponding role, then a new actor should be defined in the abuse case. For example, in our Internet-based information security laboratory, a malicious student might attempt to copy another student's solution. If we were to model this as an abuse case, we would define a new actor *Malicious Student* for the abuse case, rather than have the *Student* actor associated with the abuse case. If outsiders or unauthorized users are a threat, then new actors will

have to be added to represent them. We do this to emphasize that a different role is active during abuse, even if the abusing actor also fulfills other roles. Some customers and users can be very sensitive about discussions of possible insider threats.

Actors in use cases are defined only briefly. In an abuse case, we give a more detailed description of the actor. Actor descriptions are very useful in abuse case modeling. Three characteristics of each actor are critical to understanding an abuse case: the actor's *resources*, *skills*, and *objectives*. The third characteristic may seem redundant if our abuse cases are at the same level of abstraction as essential use cases. However, the objectives of an actor are not really captured in the abstract abuse cases. Instead, we describe the actor's objectives as long-term goals that the actor potentially seeks over more than one abuse case. For example, the abuse case model of our information security laboratory includes two actors *Script Kiddie* and *Nazgul* [9]. The Script Kiddie's objective's include demonstration of skills by breaking in to a large number of systems while the Nazgul's objectives include industrial espionage, terrorism, and war. The resources available to an actor include other persons and organizations that might assist the actor, in addition to the tools and systems the actor may be using. Finally, resources include the amount of time an actor has to devote to the abuse case.

The description of an abuse case can also slightly differ from the approach taken with use case descriptions. We can describe abuse cases in exactly the same way that we describe use cases. However, we sometimes take a different approach. A use case description centers around a single abstract transaction or sequence of events, because a use case describes a desired interaction. On the other hand, because we are not sure where flaws will occur, an abuse case describes a family of undesirable interactions. The final "implementation" of an abuse case will be through the exploitation of requirements oversights, design flaws, and implementation flaws. Since we want to use the abuse case model to reduce the number of requirements oversights and design flaws, we may choose describe many abstract "transactions" that might take place to accomplish the same abuse. Each feature or component of the target system that might be exploited in an abuse case will be considered in the abuse case description. So each security relevant feature or component in the target system adds an abstract transaction to the family.

In our limited experience, we have used a tree, or sometimes a DAG structure to describe abuse cases in this way. We use a structure that could be a sub-tree of an attack tree, as used in penetration testing. The root of the sub-tree is the system we are modeling and the leaves are the resources or components of the system that are the targets of the abuse case. The interior nodes represent subsystems, applications, and individual classes within the applications. Each path from the root to a leaf shows which subsystems, applications, and classes might be misused in order to affect the leaf node in the desired way. Multiple paths through the tree indicate alternative means of accomplishing the abuse. In our experience, the interior nodes of the tree are entities that may be regarded as subjects, while the leaf nodes are objects.

To summarize,

Use Case	*Abuse Case*
• A complete transaction between one or more actors and a system.	• A family of complete transactions between one or more actors and a system, that results in harm.
• UML-based use case diagrams.	• UML-based use case diagrams.
• Typically described using natural language.	• Typically described using natural language. A tree/DAG diagram may also be used.
	• Potentially one family member for each kind of privilege abuse and for each component that might be exploited.
	• Includes a description of the range of security privileges that may be abused.
	• Includes a description of the harm that results

from an abuse case.

In our experience, we develop the abuse case model one step behind the use case model. We use each component of the use case model to construct the corresponding component of the abuse case model:

1. *Identify the actors.* After the actors of the use case model have been identified, identify the actors of the abuse case model. If an actor in the use case model might attempt harmful use of the system, then add a corresponding malicious actor to the abuse case model. After the insider roles are represented, then actors should be added for any intruders that might be a problem. Distinguish outsiders primarily on the basis of skills and resources. Requirements documents may give some help in identifying actors for abuse cases but a careful analysis of the system environment should also be done. It is important for the security specialist to discuss the potential actors with users and customers.

2. *Identify the abuse cases.* For each actor, determine their interactions with the system. Name each abuse case. At this point, it is helpful to draw an abuse case diagram.

4. *Define abuse cases.* As the interface to the system becomes more refined and the specific components are identified, the abuse case can be described. Since we use a tree structure to describe the possible points of abuse, we defer the definition until there is enough system structure to work with. The definitions can be refined as the description of the system is refined.

3. *Check granularity.* There may be too few abuse cases or there may be too many. Deciding how many are needed is largely a matter of experience and consideration of the specific target system. In our experience there are two ways we can wind up with too many abuse cases: 1) including possible but unlikely cases, or 2) modeling with too much detail. The latter problem results in several abuse cases that are distinguished only by details that are inappropriate for the purpose. For example, in most situations we would not need two abuse cases involving password theft that differed only in the kind of command shell used to accomplish the theft. We must be cautious when discarding an abuse case as unlikely. Some abuse cases may be too complex and others may be too abstract. A good abuse case model will have uniform granularity of detail in its cases, and not too many of them. Some abuse cases may need to be refined or abstracted to achieve uniform granularity.

4. *Check completeness and minimality.* Each abuse case description should be checked to see if it describes an interaction that results in harm to a user or stakeholder in the system. We should also check to see if a critical abuse case may have been omitted. An abuse case in the model may lack an abuse based on the minimal privilege needed to accomplish the harm. Requirements documents and the use case model should be reviewed, along with descriptions of anticipated security features. Users and customers should be consulted to be sure that no critical abuse has been overlooked.

5. An Abuse Case Model For An Internet-Based Information Security Lab

An example will help to make things clear. The following example shows how we would construct an abuse case model, at the essential use case level of abstraction, for our Internet-based information security lab. We present a simplified model that is based on the actual model we developed to construct a security model and policy for our Internet-based information security laboratory.

Figure 3 shows the abuse case diagram for the lab. Our model has three actors and eight abuse cases. The first thing to notice is that we have two abuse cases that are distinguished primarily by the capabilities of the actor that interacts with our lab: *Browse Exercise with Warez* and *Browse Exercise with Scalpel*. By "warez" we mean packages and tools that allow a user to mount attacks on a system from a GUI, without knowledge of the principles involved in carrying out the attack. By "scalpel" we mean a well-engineered attack designed specifically to penetrate our system. This kind of modeling can be helpful in identifying the level of security needed by users or customers.

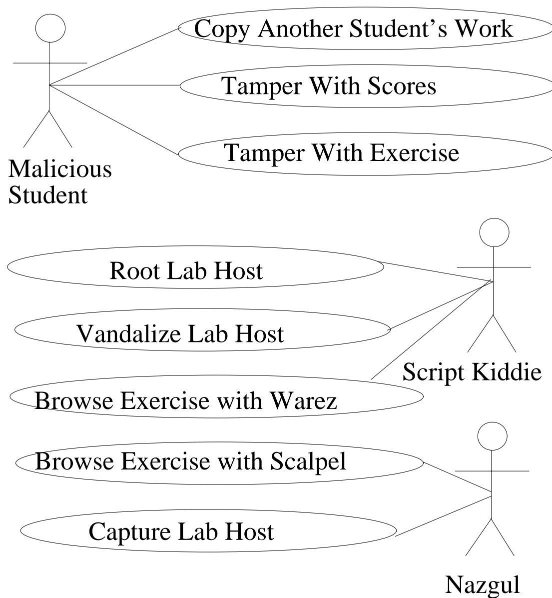

Figure 3. Abuse Case Diagram for an Internet-Based Information Security Laboratory

5.1 Actor Descriptions

Here are the actor descriptions for two of the actors in our abuse case model for the Internet-based information security lab. These two actors represent different classes of outsiders that, for whatever reason, we believe might attempt to abuse our system. The actual human who might assume the role of a Nazgul in attacking our system may also interact with it in the role of Script Kiddie, in order to probe for weaknesses. In this case, we are stating that such a person would apply no more resources, skills or time than any other attacker in the Script Kiddie role.

Script Kiddie

Resources

The Script Kiddie operates alone, although he or she may exchange some information with fellow Script Kiddies. The Script Kiddie has hardware, software, and Internet access that might be available to an individual through purchase with personal funds or by theft from an employer. Our model assumes that the Script Kiddie is willing to spend about 24 hours trying to defeat the security of a particular system.

Skills

Script Kiddies have limited technical skills. The majority of their activities are carried out using tools and techniques devised by other people.

Objectives

Script Kiddies may have a variety of criminal objectives including vandalism and theft. They also are interested in demonstrating their technical prowess.

Nazgul

Resources

Nazguls operate on behalf of groups that have budgets set aside to accomplish some form of harm. They may have technical assistance from an organization that is tasked with supporting them. They have hardware, software, tools, and Internet access provided by a business, a government, or a quasi-government. They have significant access to documentation of the systems they intend to abuse and may be able to test or simulate an intended exploit on a copy of the target system. We assume that Nazguls may spend up to 90 days in preparation and execution of an attempt to breach the security of the system.

Skills

Nazguls have superior technical skills. They can design operating systems, network protocols, computer hardware, and cryptographic algorithms. They apply software engineering technology, mathematics, computer engineering, and similar disciplines to their exploits.

Objectives

Nazguls are primarily interested in accomplishing the objective of the organization that supports them. They also seek to increase their own skills and knowledge, but not to demonstrate them to anyone. Organizations that support Nazguls do so to carry out espionage, warfare, terrorism or similar harmful activities.

5.2 An Abuse Case Description: Browse Server Exercise With Warez

Our abuse case description is intentionally very abstract, corresponding to an essential use case. We don't have the space to present a more detailed abuse case and we also want to show what an initial abuse case might look like. We would use this kind of abuse case description early in the requirements phase of a project. For example, we intend to incorporate both Windows NT and Linux based hosts in our lab and the abuse cases are meant to apply to either kind of host. However, in the requirements analysis, it is not

significant whether the abuse occurs via Windows NT or Linux. Later, in design or testing, the specific features of NT or Linux would be significant.

Notice that our description does not include the logical case where the actor (Script Kiddie) fails to gain access to the exercise materials. Since this case involves no harm, we omit it.

We have included part of a tree diagram (Figure 4) depicting the various ways that the abuse case may be accomplished. The meaning of the tree diagram is intentionally vague, to avoid complexity that is of little benefit to users. We read the diagram like a decision tree, with each path from root to leaf defining an abstract abuse case transaction that could occur. For example, Figure 4 does not show all paths of the tree corresponding to the *Browse Server Exercises With Warez* abuse case, but it does depict the complete path for an abuse that exploits vulnerabilities in the file manager of a target host, to browse the documentation for an exercise.

Browse Server Exercises With Warez

Harm: The users of the lab will be legally, ethically, and morally responsible for increasing the abilities of the Script Kiddie. The users may also be responsible for allowing information about previously unknown exploits to be released.

Privilege Range: The Script Kiddie might carry out this abuse using privileges in the following range:

1. Installation of modified system utilities with root/administrator privileges on a source or target host
2. One-time control of a root/administrator account on a source or target host
3. One-time control of a root/administrator session on a source or target host
4. Installation of modified utilities with user privileges on a source or target host
5. One-time control of a single instructor session on a server host
6. One-time control of a single student session on a server host

Abusive Interaction: Using the TCP/IP protocol suite and a hypothetical attack tool called Warez 1, the Script Kiddie requests or attempts to initiate a session on some lab host. The initial session could be on a target host, a source host, or a server host. The lab host establishes the session with the Warez 1 tool. If the initial session has sufficient privilege, then the Script Kiddie will request either a file manager, a debugger, a programming editor, or a command shell to browse exercise documentation and example exercise implementations stored on a lab server. If the initial session has sufficient privileges or there is a flaw in the system software of the host, then the lab host permits browsing of the exercises on the server host. If the initial session does not have sufficient privileges to browse exercises stored on a server, then the Script Kiddie uses additional tools Warez 2 through Warez N to request an increase in privilege. The lab machine, source, target, or server, may or may not grant an increase in privilege. If the Script Kiddie cannot obtain an increase in privilege from the system software, then the Script Kiddie requests copies of the exercise materials directly, via the available file manager, debugger, editor, or command shell of the host. One or more of these applications permit (via a flaw) browsing of exercise documentation or exercise implementations stored on a lab server.

If any of the exercise materials could serve as additional warez to the Script Kiddie, then the Script Kiddie saves or downloads them.

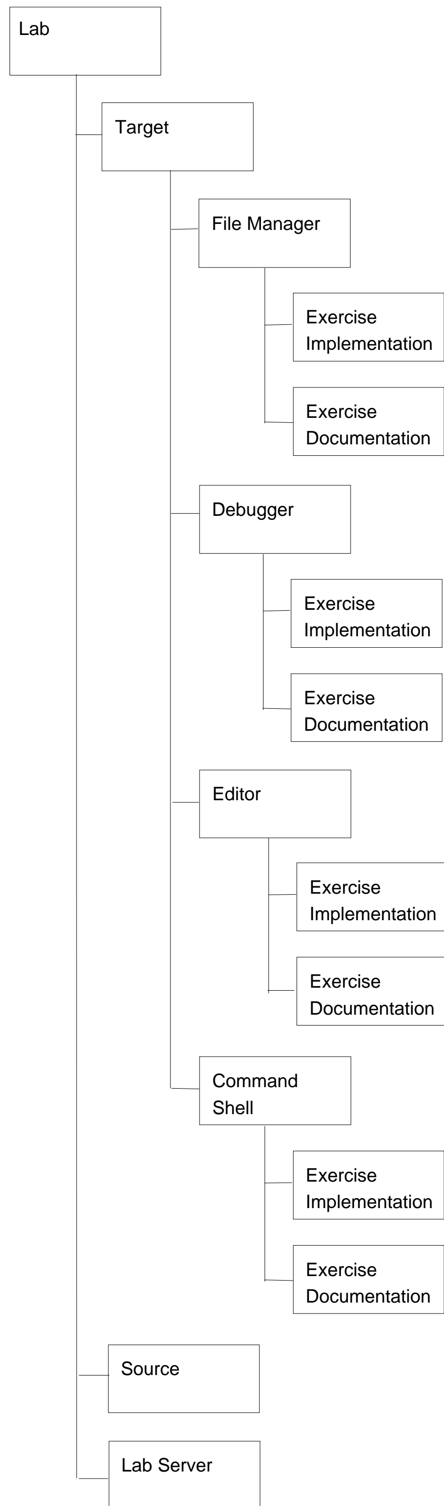

Figure 4. Tree Diagram for Abuse Case: Browse Server Exercise With Warez

The applicable abuse cases can been identified and defined in a level of detail like our example, with the help of the users and customers. Later, each abuse case can be refined and described more rigorously, as needed. One must be careful not to expend too much effort on rigorous descriptions of abuse cases early in the project. Changes in requirements or system architecture may overtake some of them and they will disappear.

6. Applications Of Abuse Case Modeling

Abuse case models can be helpful during the requirements, design, and testing phases of a security engineering process.

In a requirements phase, abuse case models can be used to increase both user and customer understanding of the security features of a proposed product. They can be made simple and abstract enough to be understood by users from a wide range of application domains. They can be used to show customers what will be prevented and what will not, in terms of their application domain. For this same reason, abuse case models are also useful for security requirements elicitation. Users can decide, in terms of their own application domain, which threats apply and which threats should be countered by product security features. Many fine security models have been developed that model various kinds of protection, in mathematically sound ways. Use of these models is essential for any product that aims at complete security. However, these models are subtle and very abstract. It can be difficult for users or customers to apply them in their own domain. Practitioners who use and translate these security models may expend a great deal of time transforming a policy to the user's domain, only to find that the policy is not what the users intended. Abuse cases may help security practitioners and users save time in arriving at a good understanding of security requirements.

During the design and testing phases of a security engineering process, we can apply abuse cases through a *refutation process*. As we analyze and design the system, we refute each use case to the appropriate level of assurance. This is one reason for describing the actors in greater detail in an abuse case. Our refutation may depend on the skills, resources, or objectives of the actor. For example, we may argue that 40-bit cryptographic keys are sufficient to refute an abuse case involving a Script Kiddie actor, because of their

specified resource limitations, but not against a Nazgul. In other instances, our refutation may be based on the properties of a design feature. The strength of the refutation can be used to characterize the assurance we have. A refutation arrived at during an informal code walk through is not as strong as a refutation based on formal methods. Abuse cases can be ranked or weighted according to the assurance that should be applied to them. The assurance budget for a project can then be allocated by abuse case, according to the ranking.

During testing, we can design our security function tests to refute abuse cases. For example, we can apply the abuse case directly as a family of test cases. We form a test team that has the same skills and resources as the actors associated with the abuse case and let them exercise our system features. We can also combine testing with other refutation arguments. We may argue that neither an editor nor a debugger can browse exercises, if the current session lacks the necessary security permissions. We can then use testing to show that all exercises are configured with the security attributes needed to prevent browsing and that all passwords are sufficiently strong. We can also rank abuse cases in order to allocate testing resources.

Abuse cases can also be used to make design trade-offs. Since both use cases and abuse cases are readily understood by users and customers, they can be used to explain security-related design trade-offs. Customers will be better informed when choosing between modified functionality in a use case and the residual risk in an unrefuted abuse case.

7. Conclusions

By borrowing the concepts and notation of a proven modeling technique, we can model significant forms of abuse that we wish to prevent. An abuse case model is easily understood not only by users and customers, but also by the many developers who understand either use case models or UML. This is a significant benefit since many developers who work on the security features of software do not understand mathematical security models. Abuse cases are also more easily understood by other project engineering personnel who are not familiar with mathematical security models.

Abuse cases are much simpler than mathematical security models but they can be an effective tool for capturing security requirements. They are particularly useful in communicating with users and customers during requirements analysis and may be easier to understand when trade-offs must be made between security and functionality.

Abuse case models are not a substitute for mathematical security models. We intentionally make abuse case models ambiguous and incomplete and do not worry about their soundness. Abuse case models do not replace any other part of a sound security engineering process. The same qualities that make them powerful in security requirements analysis render them unfit as tools for high assurance. On the other hand, we have found them to be very useful as a complementary tool, when used during the requirements, design, and testing phases of a project.

References

1. COMMON CRITERIA IMPLEMENTATION BOARD, C*ommon Criteria for Information Technology Security Evaluation, version 2.0*. May 1998, Common Criteria Project Sponsoring Organisations.

2. CUSUMANO, M. and SELBY, R. How Microsoft builds software. *CACM*, 40, 6, June 1997, pp. 53-61.

3. LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, 1998.

4. RATIONAL CORPORATION, *Rational Rose*, http://www.rational.com.

5. RUMBAUGH, J., JACOBSON I., and BOOCH, G. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

6. SANDHU, R. and MUNAWER, Q. The RRA97 model for role-base administration of role hierarchies, *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998, Phoenix, Arizona, pp. 39-49.

7. SCHNEIER, B. and MUDGE. Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP), *Proceedings of the 5th ACM Conference on Computer and Communications Security*. November 1998, San Francisco, California, pp. 132-141.

8. THOMSEN, D., O'BRIEN, D. and BOGLE, J. Role based access control framework for network enterprises, *Proceedings of the 14th Annual Computer Security Applications Conference*. December 1998, Phoenix, Arizona, pp. 50-58.

9. TOLKIEN, J. *Lord of the Rings*. Houghton Mifflin, 1974.