Abuse–Case–Based Assurance Arguments

John McDermott
US Naval Research Laboratory, Washington, DC 20375
mcdermott@itd.nrl.navy.mil

Abstract

This paper describes an extension to abuse−case−based security requirements analysis that provides a lightweight means of increasing assurance in security relevant software. The approach is adaptable to lightweight software development processes but results in a concrete and explicit assurance argument. Like abuse−case−based security requirements analysis, this approach is suitable for use in projects without security experts. When used in this way (without security experts) it will not produce as much assurance as the more traditional alternatives, but arguably give better results than ad hoc consideration of security issues.

1 Introduction

The concept of assurance is unfamiliar, if not alien, to many current software development projects. The majority of current software engineering research is focused on productivity rather than quality. In the 21st century, market forces and the inertia of accepted practice result in software produced with little specification, analysis or design. Even less attention is paid to assurance practices such as specification, review, validation, verification, or testing. While this is not good for quality in general, it is fatal for security. The security of critical but interconnected (e.g. Internet−based) systems can depend upon the security of the weakest interconnected software.

We can improve this situation with an extension to abuse−case−based security requirements analysis [10] that provides a *lightweight* (i.e. the process model has very few steps and work products, and the steps can be applied with relatively little rigor.) means of increasing assurance in security relevant software. The approach imposes fewer time and resource penalties on software development processes but results in concrete and explicit *assurance arguments* [12]. Like abuse−case−based security requirements analysis, this approach is suitable for use in projects without security experts. When used in this way (without security experts) it will not produce as much assurance as the more traditional alternatives, but arguably give better results than ad hoc consideration of security issues. It is not a high−assurance approach; it is a lightweight assurance approach.

Lightweight assurance approaches consume fewer resources than more complex or criteria−based approaches, thus they fit better in lightweight development efforts (but consequently result in less assurance). Abuse−case−based assurance arguments are intended for use in situations where a higher−assurance approach will not be used, for whatever reason. Since the assurance process structure is simple, it can be used without adding complexity to simple software development processes.

Briefly, the abuse−case−based approach produces an assurance argument as a collection of abuse case refutations. A set of candidate abuse cases is constructed at the beginning of the development project. These abuse cases are refined along with the software. Developers can construct their assurance argument by refuting each abuse case refinement. Granularity of refinement, degree of rigor, and level of assurance can be varied to suit the project.

1.1 Relationship to Other Work

Abuse−case−based assurance arguments have their roots in flaw−hypothesis penetration testing, where testing assumes the presence of a specific flaw and seeks to confirm it. In general, though not in all cases, flaw−hypothesis penetration testing takes place after systems are working. Abuse−case−based assurance arguments are applied throughout the life cycle of a software product. Flaw−hypothesis penetration testing always seeks to uncover flaws while abuse−case−based assurance arguments usually include abuse cases that are not intended to uncover flaws, but instead provide an argument that the security of the implementation is sufficient. Finally, the approach we are presenting always uses sequences of interactions between external actors and the system to describe the effects of potential flaws under investigation, rather than posing specific flaws and investigating their potential relationships. Our results are related to work on general assurance

arguments and Common Criteria.

Our work is related to software fault tree analysis (SFTA) [9] because both approaches seek to improve software quality by focusing on behavior that should not happen. Our work differs in several ways. One of the most important is that abuse cases may be used throughout the software development process while SFTA is applied after the source code is complete. A significant mathematical difference is that SFTA is based upon refinement of single undesirable events in the execution of the software while abuse−case−based assurance arguments are based upon refinements of undesirable interactions between the software and its environment, i.e. sequences. Furthermore, SFTA is intended to be used by experts while abuse−case−based assurance arguments can also be used by competent developers who are not security experts. Some more technical concepts from software fault tree analysis, such as the application of predicate transformers to undesirable postconditions, are adaptable to abuse−case−based assurance arguments.

Our work is also related to Survivable Network Analysis (SNA) [6] process, of the CERT Coordination Center. SNA uses *intrusion scenarios* which are very similar to abuse cases. SNA has been successfully applied in many practical projects and is in production use. Like our work, SNA is also intended to improve the security of systems at design time. Our work differs in that it is intended to produce an assurance argument, refines abuse cases, and uses refutations. SNA uses a *survivability map* that could correspond to an assurance argument, though it is not intended for assurance per se. The survivability map addresses architectural features whereas the refutations may be based upon architectural features or assurance measures such as verification, or both. Finally, SNA is intended for use by experts with a strong background in security engineering. Our approach could be used in this way, but is also suited for use in projects that do not have access to a security expert.

Our work is related to the attack modeling of Moore, Ellison, and Linger [11] because both abuse−case−based assurance arguments and attack modeling explicitly consider the resources and skills of the attacker. Attack modeling distinguishes different attack profiles according to attacker skills and resources. Attack profiles parameterize and generalize families of attack trees, on the basis of commonly applicable architectural reference models. Attack profiles are meant to organize and apply security failure data to the design of information systems and are intended for use by experts. Abuse−case−based assurance arguments use system−specific abuse cases to argue the assurance of an information system. Each abuse case has specific attacker skills and resources. Assurance is argued with respect to the specified attacker capabilities.

1.2 Experience

Our experience with this approach has been limited to simple projects carried out by students to construct assurance arguments for selected components of open source software. The approach has been very popular with these students, since it is more intuitive than traditional approaches exemplified by criteria−based evaluations, SFTA, SNA, attack modeling, and general assurance arguments. The students are arguably good candidates for a method aimed at competent but unspecialized developers. They have helped uncover problems in early versions of the approach and applied it in unusual ways that helped us learn more about it.

The students have been able to discover several flaws in the software (Linux PAM) that they analyzed using this method. None of the flaws they discovered could be characterized as new, because the students were restricted to working with older versions of the PAM distribution, for security reasons. This resulted in some disappointment for the students when they wanted to announce the flaws they had located, but did make the case the abuse−case−based approaches work for moderately experienced developers in poorly structured situations.

1.3 Abuse Cases

Following McDermott and Fox [10], we define an *abuse case* as a specification of a type of complete interaction between a system and one or more actors, which we call malefactors. (We allow, but do not require, the specification of well−meaning but negligent actors who may inadvertently put the product into an unsafe state.) The results of the interaction are harmful to the system, one of the actors, or one of the stakeholders in the system. A complete abuse case defines an interaction between some malefactors and the system that results in harm to a resource associated with one of the actors, one of the stakeholders, or the system itself. The malefactors in an abuse case model are the same kinds of external agents that participate in use cases but they are not the same actors. In an abuse case, we give a detailed description of each malefactor. Three characteristics of each malefactor are critical to understanding an abuse case: the malefactor's *resources*, *skills*, and *objectives*. For example, are the malefactors working alone, do they have a funding sponsor, or are they supported by a national intelligence agency? Do the malefactors have the skills to write Perl scripts? Can they write device driver code? Are the malefactors counting coup or do they have specific harm in mind?

An abuse case also describes the range of privileges that might be abused to complete the case and includes a short description of the specific harm that will occur as a result of the abuse.

In our experience, we have developed abuse case

models one step behind the corresponding use case model:

1. Identify the malefactors.
2. Identify the abuse cases.
3. Define abuse cases.
4. Check granularity, completeness, and minimality.

2 The Basic Process

Our abuse−case based assurance argument is developed by a four−step process that starts with the establishment of some boundaries on the problem:

1. Define the Assurance Problem
2. Construct the Candidate Abuse Cases
3. Refine the Candidate Abuse Cases
4. Refute the Candidate Abuse Cases

In the first step, we define the kinds of harm that the product or system is supposed to prevent and its anticipated threat environment, including a use case model and descriptions of the actors who might abuse our system. We identify and analyze any security models that are applicable (in addition to manifest models like the Typed Access Matrix model, in this step we also consider the typical "Alice and Bob" cryptographic protocol descriptions to be security models).

After the assurance problems and applicable security models have been defined, we identify and construct *candidate abuse cases*. We call these candidate abuse cases because, in an ideal argument, we intend to show that there are no actual abuse cases. We construct our candidates following the same approach we use for security requirements engineering. The last step of abuse case construction is a coverage analysis that examines the completeness, granularity, and justification of each case. Each candidate abuse case then becomes a unit of work that can be managed just like a system component, that is, we can assign cases, plan the work that goes with each one, and track our progress based on the status of the individual cases. In the next major step the candidate abuse cases are refined according to the required assurance. More refined cases yield higher assurance, but cost more to complete. In our experience, there are three levels of refinement that are useful: interface−level abuse cases, design−level abuse cases, and code−level abuse cases.

In the refutation step, we take each refinement of a candidate abuse case and argue why it cannot hold. The refinement of a candidate abuse case, together with the argument that it cannot hold, is referred to as a *refutation*. The complete set of refutations, organized by their candidate abuse cases, is our abuse−case−based assurance argument. Again, we intend to finish the process with no actual abuse cases. Figure 1 shows the basic process.

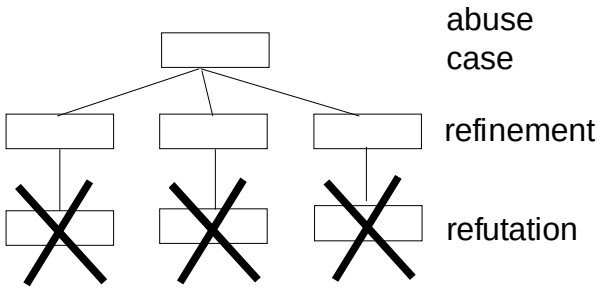

Figure 1. Basic Assurance Process

This basic process can be iterative. Discoveries made during the refinement and refutation steps can lead us back to construct new candidate cases or even to revisit the problem definition step. Sometimes, we may discover that we need new candidate abuse cases in order to reorganize our assurance argument.

2.1 Defining the Assurance Problem

One important step that we always include in the problem definition stage is an analysis and validation of all security models. The end result of this step is an assumption that all security models do what they are supposed to do (we avoid saying that the models are correct) individually and together. If the models are all taken from the literature or have otherwise been deemed well−understood, then the analysis is focused more on understanding the possible interactions between security models.

In many cases, products are developed without a specific security model. When there are no security models, we must analyze the available product or subsystem descriptions to identify and describe the de−facto models. De−facto security modeling per se is not a necessary part of abuse−case−based assurance arguments, but we have found it to be so useful in practice that we have not tried to do without it.

In both instances, well−understood or de−facto, we are concerned with analysis and validation of the security models and not their implementation. Assuring a sufficient−quality implementation of the models is accomplished through the next three steps.

It is more beneficial to define the kinds of harm to be prevented, describe the anticipated threats, and anticipate the malefactors during the requirements phases of a project. However, problem definition is an activity best overlapped past the requirements phase, into the design phases of the overall development process. The chief reason for this overlap is that some design (and implementation) features imply the possibility of certain abuses. A design decision to store large amounts of data

with confidentiality implies some form of inference attack. A design decision to use a network implies many forms of network attacks. Design and reuse decisions made during the design phase may bring us back to analyze newly introduced aspects of a de−facto security model.

If abuse case models have been used in the requirements phases of a software process, then they are incorporated into the assurance argument as part of the problem definition step and are input to the abuse case construction step.

2.2 Constructing Candidate Abuse Cases

Since refinement increases the number of descriptions associated with a candidate abuse case, we must be careful not to have too many initial descriptions. Limited experience has shown that there is a tendency to describe some candidate abuse cases in too much detail and then draw unnecessary distinctions based on the detail. During the coverage analysis, these unnecessarily distinct candidates are combined into a more abstract or less precise candidate that includes the unnecessarily detailed cases as refinements.

We also need to do some planning during this step. We must plan to match assurance resources to the overall problem. In a large assurance argument, we should also analyze the candidates to rank them according to criticality and expected effort. Some abuse cases may deserve more resources than others. Other candidate cases may be just as critical, but clearly will not result in as many refinements. Finally, we may elect to construct some candidate abuse cases that will not be refined. These cases may be revisited in future assurance work on the system or product. We may also chose to define these candidates to influence the future design or evolution of a product.

Abuse case construction activities are best performed in concert with the design phases of a software process model.

We use three heuristic classes of abuse cases to assist in the construction of candidate abuse cases for assurance arguments:

- *Model abuse*: abuse that results from failure to implement the security model
- *Bypass abuse*: abuse that results from bypassing security mechanisms
- *Tampering abuse*: abuse that results from tampering with security mechanisms

Security model abuse cases are descriptions of model failures, e.g. inability to control properly the accesses of some subjects to some objects or failure to follow the rules of a protocol for every principal. The actors participate in an interaction with the product that results in a failure of the security model itself. These kinds of abuse are possible (i.e. the abuse case is sustained) when the security mechanisms do not function as specified by the security model. For example, in Linux PAM [13, 14], the following security model abuse would be applicable:

> *Candidate Abuse Case 1:The administrator creates a configuration file which contains incorrect syntax for an authorization module. Linux−PAM parses the configuration file and does not report a problem. The incorrect syntax puts PAM into an insecure state. A person who does not have an account requests authentication with a random password and is granted access.*

We use bypass abuse cases to describe interactions where harm to the system or its stakeholders results, even though the security model did not fail. In general, this will require discussion of the context or environment of the product. Abuse where the security model itself does not fail implies the existence of (unprotected) resources outside the boundaries of the model. This requires careful attention to the scope of an assurance argument, since the product environment can be quite large or even unbounded. An example for Linux−PAM would be:

> *Candidate Abuse Case 2: A person who does not have an account requests a remote login via a service that does not use the Linux−PAM conversation correctly. The service passes incorrect data via the conversation and Linux−PAM grants access* .

For an example of an attempt to tamper, let us pick the following abuse case for Linux PAM:

> *Candidate Abuse Case 3: The unprivileged user logs into his or her normal account and runs a PAM−aware application that causes the current instance of PAM to enter an insecure state where it always authenticates any account, including root. The user then logs out of his or her normal account and logs into privileged programs, via the damaged instance of PAM.*

2.3 Adversarial or Cooperative Abuse Cases

It is important to understand the relationship between assurance argument abuse cases and penetration testing. Penetration testing may seem to be very similar. However, penetration testing seeks validated or actual abuse cases. Assurance arguments based on abuse cases also *may* contain some of these abuse cases: *adversarial abuse cases*. Adversarial abuse cases are intended to be sustainable from the start of the assurance argument. There is another kind of abuse case that is found in abuse−case−based assurance arguments: the *cooperative abuse case*. A cooperative abuse case is intended to be refuted. That is, based on our understanding of a product,

we anticipate that we will be able to refute the abuse case. (Otherwise, why build the product?) Each cooperative abuse case is present in our assurance argument to

- organize the review and analysis of the product design and code, and
- clarify the results of the assurance process.

So penetration testing does not include tests where we presume there is no problem, but abuse−case−based assurance arguments do include (cooperative) abuse cases, even when we are sure the abuse is not possible.
Our *Candidate Abuse Case 1* above might be an example of a cooperative abuse case, if we had reason to believe that the Linux−PAM parser rejects configuration files containing syntax errors. If the parser had been formally specified and verified to reject syntax errors, then we could easily include Candidate Abuse Case 1 as a cooperative abuse case. What is more likely in a lightweight software development approach is that some source code analysis of the parser would be deemed sufficient evidence.

Adversarial abuse cases are present in our assurance argument to demonstrate adversarial review and analysis of the product. The number and strength of our adversarial abuse cases indicates the amount of adversarial review that has been completed. *Candidate Abuse Case 3* might be an instance of an adversarial abuse case if we suspect that it is possible or that we lack evidence concerning the tamper resistance of PAM. Designing an abuse case as either cooperative or adversarial relates to the reason for including it, the abuse case justification.

2.4 Justification

Abuse cases are usually not an exhaustive description of the possible abuses and the refinements are likewise probably not exhaustive descriptions of particular abuse cases. Exhaustive description or refinement would defeat the purpose of lightweight assurance. Since we choose to omit some possible refinements, then each candidate abuse case and each of its refinements should be *justified*. A justification explains why the particular abuse case is worth investigating. A justification should also explain how its case or refinement contributes to the overall assurance argument. Justification can be based on the intended security models of the system or on the security policy the system will enforce. These kinds of justifications will usually be brief; no longer than the abuse case they support.

2.5 Refining Abuse Cases

Refinements add details to candidate abuse cases. These added details allow us to distinguish sub−cases to provide greater coverage and more assurance in our argument. We have worked with three levels of refinement: interface−level, design−level, and code−level refinements.

2.6 Interface Level Refinement

Interface−level refinement of an abuse case adds detail to the malefactor's inputs and the product's responses. Unlike the more elaborate design− or code−level refinements that trace through components or sequences of code, interface−level refinements refine a candidate abuse case as a black−box specification. Components and code sequences are not specified.

If we consider our first example abuse case, we see that just a few more details can produce many interface level refinements. Our first refinement adds concrete detail to the abstract notion of incorrect syntax, specifying where and how the syntax will be incorrect. (The description of module type refers to a concept that is visible at the user interface, rather than a specific module of the PAM code.)

> *Refinement 1.1: The administrator creates a configuration file which contains incorrect syntax for the* control−flag *of a module of type* auth*. Linux−PAM parses the configuration file as though the* control−flag *is the string "optional" and does not report a problem. The incorrect syntax puts PAM into an insecure state. A person who does not have an account requests authentication with a random password and is granted access.*

Our second example also specifies a more concrete notion of incorrect syntax. It also elaborates the description of PAM's response to the incorrect syntax.

> *Refinement 1.2: The administrator creates a configuration file which contains incorrect syntax for an authorization module. The incorrect syntax is intended to stack the two authentication modules pam_warn and pam_deny but a malformed module−type string (e.g. aucc instead of auth) in the second entry causes Linux−PAM to parse the second entry in the configuration file as not being an authentication module and does not report a problem. This incorrect syntax puts PAM into an insecure state, that is pam_warn will execute but access will be granted. A person who does not have an account requests authentication with a random password and is granted access after being given a warning that the service is not configured.*

Our third example shows a refinement that has a small problem. The refinement is a specific case of its parent candidate abuse case; a refinement that is distinct from Refinements 1.1. and 1.2. The concept of module

path is part of the user interface to PAM. However, Refinement 1.3 should be more specific in its description of the incorrect configuration file syntax that leads to a bad module path.

> *Refinement 1.3: The administrator creates a configuration file which contains incorrect syntax for an authorization module. The syntax error results in an incorrect module path. Linux−PAM parses the configuration file, loads the wrong module, one that will return success on very simple random passwords, and does not report a problem. A person who does not have an account requests authentication with a random password and is granted access.*

2.7 Design−Level Refinement

In a design−level refinement, we add details in terms of the modules, operations, or classes of a system, to an interface−level refinement. The descriptions of system responses are elaborated in terms of the functions or class operations that are invoked. In our limited experience, we have found it is better to work from interface−level refinements rather than try to refine a candidate abuse case directly into a design−level refinement. This allowed us to organize design−level refinements by their associated interface−level refinement. Working from interface−level abuse cases also allows us to focus our analysis on component interaction rather than dividing attention between concrete syntax and design behavior. Here is an example design−level refinement of the interface−level refinement 1.1:

> *Refinement 1.1.1: The administrator creates a configuration file which contains incorrect syntax for the* control−flag *of a module of type* auth. *The interaction between functions "_pam_parse_conf_file" and "_pam_assemble_line" parses the configuration file as though the* control−flag *is the string "optional" and does not report a problem. The incorrect syntax puts PAM into an insecure state. A person who does not have an account requests authentication with a random password and is granted access.*

We have found that some diagramming is useful in constructing design−level refinements. In our limited experience, we have used (abused, since Linux PAM does not have an object−oriented design) UML *component diagrams* [1] and also the *robustness diagrams* of Jacobson's Objectory process [7] to describe design−level abuse cases.

2.8 Code−Level Refinement

In a code−level refinement, the interface− or design−level abuse cases are elaborated into traces or symbolic execution of the relevant code. There are many potentially useful forms of code−level candidate abuse cases. We have used code−level refinements of abuse cases as sustained adversarial abuse cases. These code−level refinements served as place holders in the assurance argument, for problems discovered by the assurance team. These refinements could only be refuted by changing the design or code of the system. We have also used code−level refinements that are essentially just more detailed descriptions of design−level refinements. As an example of this kind of refinement, our interface−level Refinement 1.3 can be expanded into a code−level refinement by tracing a sequence of function calls that Linux−PAM would make to parse a configuration file. The sequence could suggest how incorrect results returned by *_pam_StrTok*, in *_pam_parse_conf_file* could cause the problem described in case Model Abuse 1.3.

The sequence of responses described by the refinement must not change the actor's inputs or responses. Following our example above, the code−level refinement could not change the actor's responses in order to cause *_pam_StrTok* to fail. However, it would be acceptable to add details about the actor's input, if that detail is necessary to trace through the code.

If a problem is discovered in the code, but it depends upon malefactor (or actor) input that is not covered by the basic candidate abuse case, then a new candidate abuse case should be added to the argument. If resources are available, it may be useful to look for other refinements of this new candidate abuse case.

2.9 Refutation

The goal of the refutation step is to discover one or more refutations for each refinement. In other words, we expect to complete the assurance process by showing that none of the candidate abuse cases hold.

We have found it useful to assume that there are two kinds of refutations: *mechanism* refutations and *assurance* refutations. In a foundational sense, the distinction may or may not be clear, but it has been a useful heuristic for us. Refutation by mechanism applies to direct policy violations: we refute the candidate abuse case by explaining how the applicable security mechanisms work to prevent the policy violation. Refutation by assurance applies to candidate abuse cases based on flaws in, bypassability of, or tampering with security mechanisms.

Refutations can be made at different confidence levels. An assurance team may decide to refute an abuse case at the interface level, without constructing any design−level refinements. This would provide less assurance than a set of design−level refinements of the same abuse case.

Refutations can be constructed by conventional reasoning techniques. The two most frequent approaches would be by counterexample and by contradiction. Abuse case counterexamples are sequences of interactions between the product and its actors, just like abuse cases (and use cases). A counterexample for a particular abuse case contains exactly the same actor events as the abuse case. However, the counterexample includes one or more responses from the product that are different from the abuse case. These distinguished responses show where the product would prevent the abuse. For a counterexample to *Model Abuse 1.1* above, we might provide

> *Counterexample −Model Abuse 1.1: The administrator creates a configuration file which contains incorrect syntax for the* control−flag *of a module of type* auth*. Linux−PAM parses the configuration file as though the* control−flag *is* ***unrecognizable*** *and* ***does*** *report a problem. The incorrect syntax puts PAM into* ***a state fail secure state****. A person who does not have an account requests authentication with a random password and is* ***denied*** *access.*

Here we might find a simple argument for our counterexample such as

Argument: Source code review of `pam_handlers.c` *and* `pam_dispatch.c` *by Ralph C. and Ed N.*

A more convincing argument would appeal to a rigorous interface specification, a trace through the sequences of execution that would be followed if the *control−flag* syntax was incorrect, a Cleanroom verification [8], or we could construct suitable postconditions and push them back through the applicable code or specifications. We omit an example of this, as it has little interest for readers who do not have the complete source code immediately at hand.

It is helpful to provide multiple refutations. Multiple refutations can be made on the basis of distinct counterexamples. This increases assurance because it is less likely that all of the counterexamples will be incorrect. Multiple refutations constructed by different parties are also helpful, since these bring distinct perspectives and experiences to bear on the abuse case.

Sometimes an abuse case is *sustained*. A sustained abuse case is a flaw in the product; an abuse case that can actually occur. Confirmation of the abuse may occur at any point in the assurance process. Once a valid abuse case has been discovered, there are two options: change the system to make refutation possible or leave the flaw in place and manage the residual risk. If we believe that we have a sustained abuse case that remains after the assurance process is complete, then we refer to it as an actual abuse case or simply as an abuse case.

3 Application

There are several issues to consider in the application of abuse−case−based assurance arguments. Abuse−case−based arguments are suitable for projects with short schedules or small assurance budgets because they are simple. Having the assurance argument organized as a collection of abuse case refutations also helps us to manage project costs during the development. Abuse cases constructed during design and development lead naturally into penetration testing. Finally, we have the skills, resources, and motives of the malefactors recorded in the candidate abuse case. This makes it easier to understand the intended strength of mechanism in the target system.

3.1 Cost Management

We can increase the assurance of our argument by creating more refinements and we can reduce the cost of our argument by creating fewer. We can also reduce cost by creating abuse case refinements but not refuting them. These abuse cases can be marked as unknown and left for future investigation or penetration testing, when more resources become available. These unknown or unrefuted abuse cases also serve as easily understandable indicators or units of residual risk. Residual risk can then be managed in a way that is more understandable to users and customers.

Abuse cases and their peer refinements are logically independent of each other. We can treat abuse case refinements: interface−, design−, or code−level, as units of a work breakdown structure. We can also treat each refutation as a work breakdown unit. This facilitates management of resources and schedule with less emphasis on the difficulty of assuring a security property (e.g. confidentiality) that may describe many abuse cases.

3.2 Integration with Penetration Testing

Penetration testing relates to abuse−case−based assurance arguments through the refutations. We can apply conventional flaw−hypothesis or attack−tree based penetration testing and specify the results as validations or refutations. We can also have the penetration testers attempt to validate an abuse case refinement from our assurance argument.

3.3 Linking Assurance to Malefactors

An abuse−case−based assurance argument makes an appeal to the skills, resources, and objectives of the attackers that are to be thwarted. While this is possible with other assurance approaches, and has certainly been considered in specific projects, it has not been done as an organized part of an assurance process. Abuse cases

provide an easy and natural means of connecting assurance (i.e., refutations) to specific classes of malefactors. The explicit connection here is that it is relatively easy to match level of effort, per refutation, and per abuse case, to the anticipated effort of the malefactor. To see how this works, suppose we anticipate a malefactor who will use a team of three people, each with an advanced technical degree and ten years of security experience, working for six months, in an attempt to leak information stored in the database part of our system, by exploiting a flaw in it. We can now discuss how many and what kind of developers will be working, for how long, to assure the frustration of the specified malefactor's attacks on the database component. We understand that matching assurance effort to malefactor effort is a complex topic with many issues that we have not addressed. Using abuse cases allows us to partition the problem, so that the complexity is limited to analysis of a specific abuse case.

3.4 Extreme Programming

Abuse−case−based assurance arguments can be useful as part of a lightweight software development approach. As an example of a lightweight approach, we choose the *extreme programming* process [3, 4]. In a world (we argue the real one) where most software is developed using a "code and fix" paradigm, lightweight process models are beneficial because they introduce some discipline where there was none before. In practice, we believe that extreme programming is arguably [2] a "least suitable" process model for developing software that has security requirements. Nevertheless, for the same reasons that lightweight process models are beneficial to chaotic software development, lightweight assurance arguments can be beneficial to assurance−poor methodologies such as extreme programming. (The reader unfamiliar with extreme programming may find it more helpful to review the materials posted at www.extremeprogramming.org than to locate the cited references.)

The assurance problem definition activities (the first step of abuse−case−based assurance arguments) can be combined with the *creation of user stories* and *release planning* activities of extreme programming.

The candidate abuse−cases can be created (the second step of abuse−case−based assurance arguments) for each user story at the beginning of each *development iteration*, at the same time iteration plan is created. During the iteration planning step, the customer (sic) can choose the number of candidate abuse cases to be considered, and also the level and number of refinements desired for each abuse case. Responsibility for constructing the abuse case refinements and refutations can be allocated to *task cards*, just like development tasks.

Abuse cases can be refined during the development of the user stories, as planned on the task cards. As software is developed to implement the user stories, the associated abuse cases can be refined as planned. (This is the third step in abuse−case−based assurance arguments.)

Each user story (viewed as a unit of work breakdown) is not considered completed until the software implementing the user story has passed all its acceptance tests. Acceptance tests are designed and implemented during a development iteration. We can add an acceptance condition to the testing activity: require a customer approved refutation for each abuse case refinement. This is the key to incorporating abuse−case−based assurance arguments into the extreme programming process model. The abuse−case refinements for each user story must be refuted before the associated software is accepted. As in other process models, acceptance tests may be used for refutations but it is not necessary to have acceptance tests for each refutation.

The following table summarizes the incorporation of abuse−case−based assurance arguments into extreme programming.

Abuse−Case−Based Assurance Argument	**Extreme Programming**
Define Assurance Problem	Release Planning
Construct Candidate Abuse Cases	Iteration Planning
Refine Candidate Abuse Cases	Development Iteration
Refute Candidate Abuse Cases	Acceptance Testing

We believe that abuse−case−based assurance arguments will never make extreme programming into an approach that produces high−assurance (or possibly even moderate assurance) software. Extreme programming is based on incomplete and ambiguous specifications called *user stories.* User stories are restricted to natural language and required to be extremely brief. However, abuse−case−assurance arguments are sufficiently simple that they can be used to gain assurance, without overburdening this lightweight approach.

4 Conclusions

Abuse−case−based assurance arguments are a lightweight approach to security assurance. They do not provide better assurance than conventional deductive assurance arguments or criteria−based evaluations. They do provide a straightforward approach for some security analysis and assurance argument in relatively unstructured projects. They can also be useful in providing some assurance to projects that lack resources

for completing criteria−based evaluations or deductive assurance arguments.

The simplicity of the approach makes it useable by technically capable developers who are not security experts. Security experts on projects that lack resources or have tight schedules can also use this approach. Beyond the inductive nature of its argument, it contains nothing that limits the current body of knowledge. To a certain extent, security experts can compensate for the inductive nature of the argument by increasing the level and quality of refinement. Where projects have limited schedules, the ability to modify an assurance argument by varying the number of refinements constructed or refuted can be useful.

The explicit discussion of each malefactor's characteristics can make both the target system's security and its companion assurance argument more understandable to users and customers. This discussion of malefactors makes assurance arguments useful for integration of systems from products and also for analyzing the configuration and operation of an installation.

Abuse−case−based assurance arguments may be a good way for engineers to apply attack profiles [11] to the design of specific systems. Attack profiles can be used to guide or justify the choice of candidate abuse cases. Attack profiles can provide a basis for understanding abuse cases that are not addressed in the design or assurance of an information system. The abuse−case−based assurance arguments can be a mechanism for relating specific security failure data to existing attack profiles.

5 Future Work

At present we are investigating three issues in abuse−case−based assurance arguments: the use of program slices [5] for code−level refinements, actual application of abuse−case−based assurance arguments to extreme programming, and matching assurance effort to malefactor effort.

We would also like to see or do some work on relating abuse−case−based arguments to certification efforts. It is not clear how this could be done for something like the Common Criteria, because incorporation of all the steps and work products would destroy the lightweight nature of the approach.

Acknowledgments

Chris Fox and the anonymous referees made notable contributions to this paper. Judy Froscher of NRL supported and encouraged this work while the author was at James Madison University.

References

1. ALHIR, S. *UML in a Nutshell*. O'Reilly, 1998.

2. ARTHUR, J., GRÖNER, M., HAYHURST, K., and HOLLOWAY, C.M. Evaluating the effectiveness of independent verification and validation. *IEEE Computer*, 32, 10, October 1999, pp. 79−83.

3. BECK, K. Extreme programming. *IEEE Computer*, 32, 10, October 1999, pp. 70−77.

4. BECK, K. *Extreme Programming Explained*. Addison−Wesley, 1999.

5. BERZINS, V. *Software Merging and Slicing*, IEEE Computer Society Press, 1995.

6. ELLISON, R. LINGER, R., LONGSTAFF, T. and MEAD, N. Survivable network system analysis: a case study. *IEEE Software*, July/August 1999, pp. 70−77.

7. JACOBSON, I. *Object−Oriented Software Engineering: A Use Case Driven Approach*, Addison−Wesley, 1992.

8. PROWELL, S., TRAMMEL, C., LINGER, R. and POORE, J. *Cleanroom Software Engineering: Technology and Process*, Addison−Wesley Longman, 1999.

9. LEVESON, N. *Safeware: System Safety and Computers*, Addison−Wesley, 1995.

10. McDERMOTT, J. and FOX, C. Using abuse−case models for security requirements analysis. *Proc. Annual Computer Security Applications Conference*, December 1999.

11. MOORE, A., ELLISON, R., and LINGER, R. *Attack Modeling for Information Security and Survivability*, CMU/SEI−2001−TN−001, March 2001.

12. MOORE, A. and PAYNE, C. Increasing assurance with literate programming techniques. *Proc. 11th Annual Conference on Computer Assurance*, June 17−21, 1996.

13. MORGAN, A. *The Linux−PAM Module Writer's Guide*, January, 2001. Linux Kernel Archives.

14. SAMAR, V. and SCHEMERS, R. *Unified Login with Pluggable Authentication Modules (PAM)*, OSF Request for Comments 88.0, October, 1995.