Experiences in Specifications:

Learning to Live With Ambiguity [*]

Mark Crosbie
Hewlett-Packard
Cupertino, CA 95014
mcrosbie@cup.hp.com

Benjamin Kuperman
CERIAS, Purdue University
West Lafayette, IN 47907
kuperman@cerias.purdue.edu

ABSTRACT

This paper describes our practical experiences in setting and working with requirements for a piece of security software. Principally, it discusses the conflicts that occurred between the ease of putting the initial requirements on paper and the difficulty in applying them. The requirements were not formally specified, but the process of turning them into code followed our standard software development process. However, the informality of the requirements was not the primary source of our conflicts; we believe that ambiguity always exists, ambiguity leads to assumptions, and assumptions are what lead to flaws – some of which may cause security vulnerabilities.

By explaining our journey through the software development process, we show how seemingly obvious and easily stated requirements lead to ambiguity, choices, and the need for revisiting specifications throughout the process. We conclude with some recommendations from our experiences that we hope will be useful to other practitioners.

1. INTRODUCTION

The intrusion detection product group at Hewlett-Packard was tasked with the design and development of a security software product that would monitor kernel and system activity to determine if there were indications of intruder or misuse activity on a system. They were acutely aware of the common mistakes that many programmers make designing and coding security sensitive software. The team was determined to avoid a repeat of these mistakes and to that end laid out a series of requirements that they felt would informally reduce their exposure to security flaws in the final product itself.

This paper discusses some of these informal requirements, how they came about, and how they were enforced in the development process. We examine how the requirements were forced to change over time both from product-release pressures and from an evolving understanding of the nature of a writing a security product. We conclude with a discussion on what lessons we learned from this experience and what recommendations we have for software architects and practitioners in industry.

The key theme of this paper is: requirements may contain ambiguity, ambiguity leads to assumptions, assumptions may lead to flaws in code, some of which may be security vulnerabilities. We will examine each phase of our product development process with respect to this theme.

2. THE EARLY DAYS – SETTING THE REQUIREMENTS

The first step in designing a security product, and one that is often overlooked, is to clearly define the threat model the product will operate against. A host-based intrusion detection system (IDS) has a particular view of a computing system which is local to itself; it does not necessarily see activity on other computing nodes. Thus, the first level of threats we ignore is those that occur on remote systems. Secondly, the data available to the IDS limits the ability to detect certain threats. As we chose to monitor kernel audit data for the first release, this precludes any network intrusion detection. Third, we decided that both trusted (i.e. root) and untrusted users would be subject to monitoring.

In determining our threat model, we focused on the exploitation of vulnerabilities rather than individual attacks. We hoped to build a product that would have a longer lifetime, and provide more useful data to a system administrator. By deciding on vulnerability exploitation detection, we were able to refine the threat model to detect the classic UNIX exploits.

A requirement that fell out from the enumeration of our threat model was that of the "best-effort principle": if the system is compromised the IDS will do its best at providing an alert. After that, the user should not trust further alerts that come from the IDS on this host. While this would seem to be self-evident, it turned out to be a powerful principle that allowed us to choose between design choices and coding options. In addition it defined the boundaries of the product's operation and interaction with its environment.

With the threat model and its "best-effort principle" in

[*]Portions of this work were supported by the Hewlett-Packard Company and other sponsors of CERIAS.

From the proceedings of the First Symposium on Requirements Engineering for Information Security, March 5-6, 2001, Indianapolis, Indiana.

CERIAS TR 2001-18

place, the team felt confident that they could draw up a set of requirements that would guide development.

At the start of the product development, the lead architect set out some basic requirements for the design of the product based on his experience in studying software vulnerabilities, flaws discovered in other products, and knowledge of research within the security community. The requirements were the following:

1. It must not require privilege to operate.
2. It will limit environmental assumptions.
3. It must not attempt to "guess" the right action - if something looks incorrect, stop and tell someone.
4. It must behave in a fail-stop manner.
5. It must tolerate failures of its subcomponents.
6. The overall security of a system should not be diminished by the installation of this product.

We had the following developmental process requirements as part of the corporate lab's development standards:

7. Our code must not exhibit common security coding flaws.
8. All code will be subject to a formal team code-review process to identify and flag errors.
9. Automated test cases and code-coverage tools will be used to ensure that code has been subjected to adequate testing.
10. The code shall make minimal assumptions about the resources on the underlying system.

We attempted to specify these requirements in a platform independent manner. For example, the first requirement states that the product should not require privilege to operate, however, it avoids defining exactly what "privilege" means for a given platform. We will discuss these points briefly to explain the reasoning behind them. We also discuss our discovery that no matter how solid the initial set of requirements are, one should plan to encounter pressure to change them in unexpected ways as development proceeds.

Operate without special privilege

Software faults that occur in executable code running at a level of heightened privilege can expose a system to greater risks (in general) [13]. To minimize the amount of exposure, it is considered good practice [3, 8, 9, 14] to minimize the amount of time that code runs in this state. Therefore, we decided to require that the intrusion detection system run without enhanced privileges.

Limit environmental assumptions

Many software faults occur because a programmer made assumptions about the run-time environment [2, 6, 7, 9]. We therefore required the product to verify all environment variables, paths and file ownerships before using them. If needed, it was required to sanitize environment variables before using them.

Do not guess

If the product could not determine the correct course of action it would not attempt to deduce it. It would simply stop with an error message. For example, if it could not locate a required configuration file, it would not attempt to deduce the configuration values.

Fail-stop

The basic premise of *fail-stop* behavior is that to prevent exposure to vulnerabilities, a protocol will examine all inputs and abort the transaction in the event of unexpected data. If errors did occur, the product was to stop execution with an error message [10].

Tolerate subcomponent failure

The product must continue to operate in a recoverable manner if sub-components failed. Perhaps a child process experienced segmentation violation due to a software flaw, exceeded some resource limit, or was killed by an administrator. The product must anticipate such failures and have procedures for handling these events. Over time it became obvious that this requirement would conflict with requirements 3 and 4 outlined above.

Security must not be weakened

As the goal of the IDS product was to *improve* the state of security on the system, an obvious requirement was that the IDS should not degrade overall system security. This meant that we must strive to minimize the additional risk exposure that our product incurred.

3. THE DESIGN PHASE

As expected, the initial product design was guided by the requirements. The design called for a series of cooperating processes that assumed they were operating in an untrusted environment. Most of the design work focused on the protocol describing how these processes would interact, with a keen eye to how these interactions could be used by a knowledgeable attacker against the product. At the design phase, many of the requirements were taken literally; the fail-stop requirements led to a design that would not progress executing beyond the point at which a security failure was detected. As a result, the design failed to take into account product component recovery, for example restarting a failed child process. We discuss each of the requirements below and how they were viewed in the design phase.

Operate without special privilege

The first instance of an ambiguity in the requirements was found during the design phase when deciding with what privilege the product should operate. Requiring that the product not operate with privilege implies that it must operate as a non-privileged user. We decided we could best meet this requirement by creating an "IDS" user to own the software and executing processes, and this user identity would be created as needed during installation. Such a seemingly simple operation turned out to have many complexities, especially in environments that operated with NIS or shadow passwords. The design phase had to address the issues raised by adding a non-privileged user. Many third-party software components assume that they will be installed and run with "root" privilege, and we had to verify that any code we incorporated met this requirement.

We discovered that the straightforward requirement "operate as a non-privileged user" contained an unstated assumption; namely that all of the data we needed to collect would be available to a user without needing special privileges. In order to collect data on failed attempts to log into the system, we discovered that we had to break this requirement.

Login attempts occur within application level processes owned by root. The kernel does not normally see failed login attempts, but they are reported through the syslog facility and stored in a file named `btmp`, which is only readable only by root for security reasons. This left us with the following four options:

1. Do not collect this data.
2. Modify the permissions on the file to be readable to our user.
3. Modify either all login programs or syslogd to allow us to collect the data.
4. Have a process with permissions to read that file.

The first option violated our non-security functionality requirements. To handle log rollover, the second would require us to have a mechanism to re-apply the permission changes which would likely require enhanced privileges. The third was unlikely from an organizational perspective – those programs were owned by other development groups and changes could only be made with their approval. The fourth solution met our needs and appeared to have an acceptable amount of deviation from our security requirements.

Limit environmental assumptions

Environmental assumptions remain hidden in every design, no matter how hard the designer tries to make them explicit. For example, a design that assumes it can create files, assumes that it is running on a writable media file-system that has enough space remaining to create the required file.

Sanitizing the environment was implemented on the "everything that is not expressly permitted is denied" principle. We chose sane values for various shell environment variables and decided that no user supplied variables would be honored. All return values from standard library and system calls were checked and handled as appropriately as possible.

Despite our best efforts, we feel that the assumptions we made still have left room for flaws – we don't believe any program of this size can be guaranteed to be error free. These flaws may express themselves as vulnerabilities, although we hope to detect and correct many of these before they affect customers.

Do not guess

The "no guessing" requirement was easy to satisfy in the design phase; any failure caused an error message to be generated and the program returned to a stable state.

Fail-stop

Coupled with "limit environmental assumptions" and "do not guess" characteristics, this requirement involved shutting down processes whenever an error or unexpected transition occurred.

Tolerate subcomponent failure

To tolerate subcomponent failure, we first needed to anticipate and plan on each module or function being unable to complete their task. This involves the design of meaningful return codes from subroutines, and appropriate response routines in the invoking programs.

There was a certain degree of ambiguity in this requirement, especially when it is taken with the "fail-stop" and "do not guess" requirements. For example, how many failures should the design allow before deciding that a non-recoverable error has occurred? Can recoverable errors be distinguished from unrecoverable errors (e.g. an interrupted system call vs. an out-of-memory condition)? We made certain assumptions in response to these and other questions. Despite our best efforts, these assumptions may contain security flaws – failure to operate when disk space is exhausted, for example. By being able to discern the ambiguity that existed between the requirements, we were able to focus our efforts on clarifying areas of confusion instead of spending that time on detailing the better understood aspects.

Security must not be weakened

Despite being a laudable requirement, we discovered that this requirement was difficult to satisfy in the design phase. Part of the difficulty arose because it is hard to quantify what a "reduction in security" is. The team had a conceptual understanding of this requirement, don't make security mistakes when programming, but turning this into a design was difficult.

To satisfy this requirement, we examined what steps one would take to weaken the security of the system. For our definition of weaken, we chose "introduce new vulnerabilities" – the IDS product should not provide new avenues for attack into the system. We examined how other products had introduced new vulnerabilities and determined how we could avoid these through careful design and coding. Many of these preventive measures were already captured in the other security requirements, but this rule became very useful for evaluating design proposals and modifications. For example, a kernel change that met all other requirements but might expose kernel data tables to user modification would not meet this requirement and therefore should not be permitted.

4. THE CODING PHASE

The requirements for the product which we had laid out earlier were seriously challenged in the coding phase. During this phase, many assumptions hidden in seemingly obvious requirements were exposed. Below, we discuss how the coding phase forced us to re-evaluate many of our basic requirements.

Operate without special privilege

Despite the requirement that the product operate without any elevated privileges, we were forced to compromise in coding. We limited the duration that the product executes with elevated privilege according to the principle of least privilege [3, 8, 14]. We knew up front that this requirement might be difficult to follow stringently depending on the data sources employed, but we had hoped that any transgressions of this rule could be removed before a final release[1].

Limiting environmental assumptions
Programmers make implicit assumptions about sizes and limits when they write code. Breaking these unwritten assumptions is the key to a type of vulnerability exploitation. Unfortunately, we were forced to make environmental assumptions. (Many were arbitrary decisions that had to be made without sufficient data to know the "right" answer.) It was difficult to know what assumptions to make, because the requirements were too nebulous to guide us. For example, how long should we wait for a process to terminate? How many times should you try to restart a failing child process? Should you grab all the memory you will ever need at start up so `malloc()` can never fail? Requirements of survivability in the face of resource exhaustion are easy to state, but the implementation of solutions in code is very difficult. We recognize it is likely that there are hidden security flaws within these assumptions. We continue to utilize our software engineering processes to find and correct these flaws.

Integration of modules
The integration of modules authored by two programmers often allowed us to discover as-yet undiscovered ambiguities within our specifications. Frequently, this occurred when two modules were being connected. Individual engineers utilized their unique coding styles when solving problems, and we failed to document some of the assumptions which they made. The same engineer who designed a module would then write the code for that module, creating a situation where the assumptions made were not violated within the module, but broke down when two modules were integrated.

One example of this occurred between two modules, one that read in data and another that formatted it. The formatting module assumed that it received a null terminated string from the input module and used that assumption throughout. However, the author of the reader module passed only the data bytes, not the termination character. This coding flaw descended from an inherent assumption in the design (all strings should be null terminated), and our security requirement of maintaining careful watch over the length of strings. The result was two implementations based on differing assumptions.

The ambiguity was in the definition of the length of a string – is it the number of characters of data, or data plus terminator. Each module contained a differing assumption as to how this ambiguity was to be interpreted, leading to software flaws which (if undiscovered) might have been vulnerable to attack. Resolving this ambiguity required us to clarify our assumptions, recode the modules appropriately and review all of our code for other instances of this assumption.

Do not weaken overall system security
As discussed previously, we did not want to add new avenues for attack against the system. If the same security problem occurred more than once we decided to solve it once and solve it well, and then reuse the solution. For example, we found ourselves continually worrying about opening and creating files in a secure fashion. This led to the creation of a library of secure implementations of common functions. We created `secure_tmpfile()` and `secure_exec()` to enforce the security policies enumerated within our design.

We continued to find subtle bugs in this library, despite many hours of code review spread over months of work by several engineers! Each pass through brought new errors to light. Some were coding flaws, but most were violations of our security requirements (subtle race conditions, buffer size assumptions, environment assumptions, etc.). Our secure library is a microcosm example of a situation of how specifying requirements is easy but the implementation of them is difficult.

[1]This was not able to be removed before the initial release and remains on the list of goals for future development.

5. TESTING, AND BETA FEEDBACK

Once the code reached a level of stability, we focussed on integration testing and product usability. It was at this time that our initial requirements were subjected to a detailed scrutiny in the operational arena. One of the initial design requirements was that the product exhibit fail-stop behavior and halt its execution when a serious error occurred. Over time, it became apparent that a product that behaved in this manner would be of limited utility to our customers. In fact, our product had to survive in a hostile environment, halting operation at the first sign of trouble was an unrealistic operational choice. To address this, we decided to modify our design and recode some key components to support failure recovery. If a child process failed, there would be some attempt to restart it.

While this appears to be a reasonable requirement at first glance ("restart processes if possible when they fail"), the implementation exposed many conflicting issues: how many times should we attempt to restart? If we fail in those attempts, should we then halt operations? Can the product continue to "limp along" without a key component operating? Will we break the "no guessing, no assumptions" requirement by operating in a recovery mode? These questions were addressed by revisiting and revising our initial design specification over a course of meetings. The formal review did not catch all of the instances in which this conflict occurred, but those were addressed informally whenever detected.

6. WHY DID OUR REQUIREMENTS CHANGE DURING THE SOFTWARE DEVELOPMENT PROCESS?

As our team moved through the software engineering process from design into final implementation, we encountered various ambiguities hidden within our initial requirements. The need to re-examine and change the requirements resulted from the following:

- Personal processes – Each contributor to the project also brought their own personal view of how requirements should be handled.

- Internal developmental pressures – As we moved through the design and coding phases we changed our notion of how the requirements were interpreted.
- External pressures – The typical software product schedule is seldom realistic. Despite our best efforts to track and plan our development process, we found that schedule slip occurred because of external dependencies.

We examine some of these pressures and how they forced changes in our requirements.

Skills problem

Security coding demands a high degree of paranoia and attention to small details. A general requirement such as "no additional vulnerabilities" leads to some very detailed coding scenarios. Not every programmer can write code to this level because of their background, past experience in writing robust software, or education[2]. Mentoring and education within a group is needed. Given the rate of employee turnover, mentoring consumed more time than originally anticipated.

When requirements are set, it is usually assumed that every engineer can understand, internalize and code to the requirements. Unfortunately, such an idealized engineer does not exist. A formally rigorous requirements phase has little value if the engineers do not have the skills needed to translate the requirements into solid code.

It could be argued that the initial requirements were poorly specified. Certainly, the product team went through no formal requirements specification process. Nor did they have the time or skills to do so. The initial requirements were internalized by the development team and provided the guiding compass for development choices. We began to question the requirements as evidence for their unsuitability to the product target environment began to mount. We were forced to revisit the requirements during testing whenever the product's operation was deemed unsatisfactory.

Failure to document ambiguity

Even though we recognized that the requirements were ambiguous, we sometimes failed to document the ambiguity as it was found. Furthermore, our decisions to remove any ambiguity should have been documented.

Code from outside parties

To speed up the development time, certain portions of the code were obtained from other groups within HP. We also believed that development time could be reduced by utilizing third party code produced by experts in a particular field (e.g. encryption). Our assumption was that such code would be a close match to our pedantically paranoid coding requirements. In cases where this assumption did not hold, we were forced to balance our production schedule against the severity of disparity and the time needed to rectify the situation.

In some instances, externally developed code did not meet our specified design standards for the overall project. It might not anticipate failure, not perform in a fail-stop manner, or even contain well known (though still common) programming flaws. We are unsure if we saved any significant amount of time by not having to author the code ourselves as we needed to expend a fair amount of resources incorporating the outside code into our stated design framework. Much of the problems in this category occurred in previously developed commercial packages or licensable security libraries that could not be customized due to our internal management, schedule, or financial pressures.

Review meetings uncover ambiguity

We held reviews of the product design to verify that it met our customer requirements. We also held code review meetings during the development stage (we define a "code review" as a formal inspection of the code by a subset of the entire team). The goals are to log the defects found and to educate the author and the rest of the team about the problems noted. Despite the effectiveness of these meetings at uncovering many errors, the review process cannot guarantee the discovery of all errors. For example, the secure code library was reviewed three times over a period of six months. During each iteration, a new, subtle bug was found. In fact, it became a right-of-passage within the team for new employees to hone their security skills by attempting to find new defects in the already-reviewed library code! Some of the defects found resulted from ambiguous requirements such as "no additional vulnerabilities." For example, if the Unix OS makes it very difficult to securely delete files, should we count that as a defect in our requirements, or in our implementation? However, the review process provided an excellent opportunity to uncover hidden assumptions and inter-module problems. It also provided a non-judgemental forum in which engineers could question requirements and agitate for change if they felt that the requirements were not justifiable.

Software metrics

One of the internal lab requirements was the use of various code coverage techniques. Depending on the technique, a certain minimum level of coverage percentage was required during the testing phase. However, our requirements of not making assumptions led to the creation of code paths that, theoretically, could not be reached. For instance, a flag is checked, and then reset – but shortly thereafter another set of tests are made, one of which is based on the value of that flag. The lab metrics pressured us to remove such checks and paths, while our requirement against making assumptions indicated it should remain.

Conflict between security and usability

We feel that a key reason the requirements were forced to change over time was the conflict between security and usability. A contributing factor to the change was the problems encountered in turning high-level

[2]How many computer programming courses emphasize the need to validate input data? How many are still accepting the use of `scanf()` for input or `strcpy()` to duplicate data despite the number of reported vulnerabilities these create?

requirements into secure code. Despite the requirement that environmental assumptions be limited, it is inevitable that the code produced will contain subtle assumptions.

Ease-of-use requirements often conflict with a straightforward implementation of security requirements[3]. Of course, the definition of "easy to use" varies greatly with the level of expertise of the user. We generated information on the ease of use from feedback provided by our alpha/beta testers, usability testing, and documentation review from people outside of the programming team.

As part of our installation, keys must be generated to encrypt communications between our GUI and the individual agent detecting intrusions. Our initial design called for a four step procedure involving two transfers of information between the GUI and each individual agent. Due to user feedback, we re-examined the protocol. We managed to simplify this to a three step process requiring only one transfer without a reduction in security.

However, the transfer step still must be performed using a "secure channel" (a nebulous term which we explained through the use of examples such as carrying a floppy disk between machines or via an encrypted network connection). The likelihood of customers obeying the intent of our warnings to not transfer certificates via email is expected to be low; however, we could not justify the substantial decrease in usability that would be required to attempt to enforce this requirement is followed by the end user in a manner that we had specified. The ambiguity of our requirement for a "secure channel" might be misinterpreted (or ignored) by a user exposing them to a potential threat.

7. RECOMMENDATIONS

In some ways ambiguity can be viewed as being both positive and negative. While it is frustrating to have to re-hash design decisions throughout the lifecycle of software development, requirements that are clear and easy to understand in a general context lead to better understanding of the requirements. The compactness of these understandable (but sometimes ambiguous) requirements are able to be kept in mind by all the participants, and by noticing where the ambiguity or conflict between rules exist, more energy can be focussed on these "hot spots."

Balance explicitness against platform independence

It may be tempting to set down explicit and detailed requirements and hope that ambiguity will be removed from the development process. We feel that overly detailed requirements specification will cause a proliferation of documents; one for each target platform of the product. There is a greater chance that these documents will become out-of-sync with each other.

To achieve the required balance we recommend that practitioners are aware of the general principles of security and build their requirements from there. For example, a requirement that states "will not create world-writable files" is preferable to "will not create mode 666 files." However, there are times when platform details need to be exposed. We feel that these details should be minimized in a high-level requirements document, and that reference should be made to a per-platform assumptions document.

[3]It seems that ease-of-coding and ease-of-use are often inversely related.

Maintain an "assumptions document"

Assumptions are similar to the pre- and post-conditions on procedural entry and exit points. Documenting such assumptions during the lifetime of the project is a valuable task. However, assumptions span far more than procedures in code: they can also include coding standards, product operations, security goals and organizational philosophies.

The assumptions document must be kept current. It is essential in the face of inevitable employee turnover and will become a valuable educational tool over time. In addition, it captures the thinking of many individuals in one place and helps offset the skills problem identified earlier.

What do we mean by assumptions? Every time a developer faces a decision in design or coding forced by an ambiguous high-level requirement, the choice made, and the reasoning why it was made, should be recorded in the requirements document. In addition, if a high-level requirement must be satisfied in different ways for different platforms, the decisions made should be recorded.

While capturing this level of detail would be wonderful, the reality is that if every small decision made during design or coding were to be reflected in an assumptions document, the development process would quickly be bogged down in administrative paperwork. To minimize the amount of overhead imposed, we suggest using the following questions as triggers to the developer to update the assumptions document:

Are hard-coded values used?

Are any magic numbers, hard-coded error messages, or file names defined and/or assumed? Are upper or lower limits assumed for boundary conditions? Any occurrence of these should be documented.

How are objects created, deleted or modified?

We loosely define "objects" to be any system entity that may have a system-wide impact. For example, a file, a process or a semaphore are objects by this definition as they are visible to other system entities. When considering an action that affects these objects, a developer must consider two criteria:

1. Will other system entities notice an adverse change after the action is taken? Will the action cause a reduction in available system resources for other processes? If so, the assumptions made about the impacts of the action must be documented. For example, if all the semaphores in the kernel are allocated, will other activity on the system be adversely affected?

2. If another entity changes an object, will the action need to be modified? This criteria is the same as the previous one but from the other perspective – do the assumptions rely on an unchanging underlying system, or do they still hold if the system state is constantly changing.

What object properties are relied upon?
If the developer is relying on an object property to be constant, then that assumption must be documented. For example, is the developer relying on the file system to always have enough space for a write? Does a process have sufficient privilege to create another process on the system?

What timing assumptions are made?
Does the ordering of actions matter, and if so, what assumptions have been made about ordering? If there are cooperative processes, does the sequence in which they are awakened matter?

Is atomicity of action assumed?
Can the action on a system object be assumed to be atomic, or can other unforeseen actions be interleaved with the current one and cause unpredictable results? If the developer is making an assumption about atomicity of action then that must be documented. For example, a common programming mistake in Unix is to check for the existence of a file using the `stat` system call and if the file does not exist, to create it using the `creat` call. While the programmer may assume that these two steps can be considered atomic, in reality an attacker may create a symbolic link in the interval between the `stat` and `creat` calls, leading to a classic race-condition exploit. Had the developer documented this assumption of atomicity, it is more likely that the potential race-condition would have been discovered and fixed by a team review.

Recognize that the requirements will not be complete

The more general the security requirements, the larger the set of implementations which will fit these requirements. Inversely, the more specific the requirements, the smaller range of possible implementations or interpretations. However, the specifications cannot always be fully defined at project inception, and their revising needs to be a continual part of the software development process. Ambiguities will always exist between the requirements, design, and implementation components of a project.

Incorporation of outside code into your project requirements requires the *reconciliation* between two teams' sets of security requirements. Even if one team were able to dictate the initial requirements, the interpretations of the software engineers in the other group would likely differ. In order to unite everything under one common set of requirements, these differences must be uncovered, examined, and unified.

The environment in which a product will ultimately operate must be factored into the requirements specification. Failure to do this will result in uncontrolled environmental assumptions appearing in the code. Anticipating these assumptions at the requirements stage will help control and track them; leaving them to the coding stage results in different assumptions for each author's assigned subset of code. It is important to understand both the deployment environment and threat model when codifying security requirements.

Recognize that requirements may change
We began with our set of security requirements when the initial design and specifications were made. In the course of development, we were forced to compromise on a few issues. We allowed "fail-stop" behavior to become "fail-continue" in certain situations. Additionally, our requirement of "no assumptions" transformed into a best-effort operation for the sake of usability. Requirements specification will seldom anticipate every circumstance or situation that will be discovered during the development cycle, so having a mechanism by which ambiguity in requirements can be revisited and the specifications revised is essential. For example, ease of use concerns, or other external pressures, might dictate a change in protocol. Originally, this product used a four step process with three secure data transfers. After initial user testing was performed, this was an area that was considered too cumbersome for commercial release. After some reworking of the protocol, we were able to produce an equivalent three step procedure that only required one secure transfer.

People must turn requirements into code
To successfully utilize security requirements as part of a design, a team needs a requirements "guru" to get the initial specifications as correct as possible. The guru must mentor the development team and explain why and where these requirements are needed. Individual engineers may not have the skills needed to adequately translate a requirement into a secure, functional piece of code. If requirements appear to be nebulous and capricious, then they will not guide the engineers and might be ignored by them during the development process. Even though change is often necessary, people have great difficulty in changing their perspective once they gain the understanding of a set of requirements.

By adopting (or at least considering) the above recommendations, we believe that a software development team can define and utilize a manageable set of security requirements. The requirements will be phrased in a manner that can be easily understood by attempting to remain both simple and platform independent. The development process will anticipate the need to educate programmers, review code, and revisit the requirements. Ambiguity and conflict between security requirements should be expected, and through proper communication and planning these issues can be addressed and the resolutions recorded.

8. ADDENDUM
Neither of the authors of this paper were knowledgeable in the academic field of requirements engineering during the development of this project, nor the initial writing of this paper. Hewlett-Packard utilizes a well defined, mature

software development process which we followed as part of our software engineering procedure. This process utilizes many tools and techniques to refine the requirements of a project including the following: validation of sponsor's intentions; balancing schedule, scope, and resources; flexibility matrices; dependency diagrams; work breakdown structures; critical path analysis; logical dependecy relationships; design meetings; Is/Is Not documents; and others. We felt that our experiences as practitioners might be valuable to those building new solutions in the requirements engineering arena.

Our reviewers pointed us towards Axel van Lamsweerde's ICSE 2000 publication [15] which presents an overview of the past 25 years of academic research in Requirements Engineering. Reading through Lamsweerde's paper, it is clear that some of the discoveries within the Requirements Engineering community are incorporated into our software development process. Reading his historical discussion we see that many of the items we encountered are well known within the academic community. Bell and Thayer's 1976 paper [4] discusses ambiguity in requirements and the need for continuing review and revision during the software development process. Ross and Schoman 1977 paper [12] discussed the need to define the environment in which a system would be developed and deployed, and that such things need to be considered in the construction requirements. Other techniques such as domain analysis, elicitation, negotiation and agreement, basing requirements on goals ([1, 5, 11] and others), iteration, and revision are described in [15] and seem similar to the processes we followed. We believe that the problems and solutions documented in requirements engineering research are reflected by our experiences. Our exposure to some of these more formal techniques will prove valuable as we continue our software engineering.

9. REFERENCES

[1] A. I. Antón. Goal-based requirements analysis. In *Second IEEE International Conference on Requirements Engineering (ICRE '96)*, pages 136–144, Colorado Springs, Colorado, 15–18 April 1996.

[2] T. Aslam, I. Krsul, and E. H. Spafford. Use of a taxonomy of security faults. Technical Report TR-96-051, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, September 1996.

[3] AUSCERT. *A Lab engineers check list for writing secure Unix code*, rev.3c edition, May 1996.

[4] T. E. Bell and T. Thayer. Software requirements: Are they really a problem? In *Proceedings of the 2nd International Conference on on Software Engineering*, pages 61–68, San Francisco, USA, 1976.

[5] V. Berzins and Luki. *Software Engineering with Abstractions*. Addison-Wesley, 1991.

[6] M. Bishop. A taxonomy of unix system and network vulnerabilities. Technical Report CSE-95-10, University of California at Davis, Department of Computer Science, University of California at Davis, Davis, CA 95616-8562, May 1995.

[7] M. Bishop and D. Bailey. A critical analysis of vulnerability taxonomies. Technical Report CSE-96-11, University of California at Davis, September 1995.

[8] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[9] S. Garfinkel and G. Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991.

[10] L. Gong and P. Syverson. Fail-stop protocols: An approach to designing secure protocols. In R. K. Iyer, M. Morganti, F. W. K, and V. Gligor, editors, *Dependable Computing for Critical Applications*, pages 79–100. IEEE Computer Society, 1998.

[11] G. F. Hice, W. S. Turner, and L. F. Cashwell. *System Development Methodology*. North-Holland, Amsterdam, 1974.

[12] D. T. Ross and K. E. Schoman. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, January 1977.

[13] R. S. Ross and T. R. Malarkey. Integrity in Automated Information Systems. Technical Report 79-91, National Computer Security Center, Sept. 1991.

[14] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, Mar. 1972.

[15] A. van Lamsweerde. Requirements enignieering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on on Software Engineering*, pages 5–19. Association for Computing Machinery, ACM Press, June 2000.