Trike v.1 Methodology Document [Draft]

Paul Saitta[*], Brenda Larcom[†], and Michael Eddington[‡]

July 13th, 2005

Abstract

Trike is a unified conceptual framework for security auditing from a risk management perspective through the generation of threat models in a reliable, repeatable manner. A security auditing team can use it to completely and accurately describe the security characteristics of a system from its high-level architecture to its low-level implementation details. Trike also enables communication among security team members and between security teams and other stakeholders by providing a consistent conceptual framework. This document describes the current version of the methodology (currently under heavy development) in sufficient detail to allow its use. In addition to detail on the threat model itself (including automatic threat generation and attack graphs), we cover the two models used in its generation, namely the requirements model and the implementation model, along with notes on risk analysis and work flows. The final version of this paper will include a fully worked example for the entire process. Trike is distinguished from other threat modeling methodologies by the high levels of automation possible within the system, the defensive perspective of the system, and the degree of formalism present in the methodology. Portions of this methodology are currently experimental; as they have not been fully tested against real systems, care should be exercised when using them.

Contents

[*]pbs@dymaxion.org
[†]asparagi@hhhh.org
[‡]michael.eddington@ioactive.com

Introduction

Trike is a unified conceptual framework for security auditing from a risk management perspective through the generation of threat models, with an associated tool which is currently under heavy development. This document describes a stable version of Trike to allow the community to start using the methodology while it's still under development. Many things here will change, if they haven't already, but we feel it's more important to release now and get a usable methodology out there.

We approach threat modeling and indeed all system auditing activities from a risk management perspective. It is impossible to completely secure any system against all attackers, and thus we are charged with ensuring that the countermeasures against attacks are appropriate given the risks of those attacks which they defend against, and the efficacy of those countermeasures. In generating a threat model, we attempt to do the following four things:

1. With assistance from the system stakeholders, ensure that the risk this system entails to each asset is acceptable to all stakeholders.
2. Be able to tell whether we have done this.
3. Communicate what we've done and its effects to the stakeholders.
4. Empower stakeholders to understand and reduce the risks to themselves and other stakeholders implied by their actions within their domains.

A threat model is a systemic and systematic evaluation of the security risks of a system. It must examine all potential risks throughout the system and not concentrate only on where holes are expected to be found, and it must evaluate the security of the system as a whole, as opposed to only looking at the integrity of individual pieces. This analysis must be performed in as systematic a manner as possible, to ensure correctness and completeness. Threat models are useful in finding holes at both the business logic and architectural levels, and can be used to organize and drive the entire security process, ensuring the completeness of analysis at the implementation level. Design flaws, conflicting requirements, unexpectedly exposed interfaces, lapses in policy enforcement, and misplaced trust can all be very hard to find without a threat model — when these issues are found without one, it's mostly a matter of luck.

All security analysis work, including threat modeling, requires trained security experts. However, much of the work in threat modeling can and should be automated, allowing the experts to focus their time and attention where it is required. The formalisms in the Trike methodology are designed to support automation to the greatest degree possible. These same formalisms also allow us to give strong guarantees which other, more ad-hoc methodologies cannot; specifically, that when we enumerate all threats against an application, we have in fact enumerated all possible threats. If the attack library used is complete, and the implementation model correct, all currently known attack methods have been matched against those places where they could succeed. Furthermore, if the formal model of the application is constructed correctly, multiple independent auditors will arrive at the same conclusions. This allows applications to be re-audited by different teams at different times while arriving at compatible results, and for the comparison of different applications.

Beyond its more systematic methodology, Trike differs in focus from other existing approaches to threat modeling in that it focuses on modeling threats from a defensive perspective, not that of an attacker. We begin by understanding the application itself, in the context in which it is used. Without a thorough, formal understanding of the application, it is impossible to examine the threats to the application. We also separate technology-specific issues from application-specific issues, making IP reuse easier between applications. Further, we separate implementation issues from requirements and business needs issues, allowing conflicting requirements to be easily identified. The automation inherent in Trike allows for quicker results from less initial information, and much more complete results with the same amount of effort, as compared to other methodologies. Trike's multiple models work together to allow the amount of information gathered about different areas of a system to be tailored to business needs and different areas of a system to be examined in differing levels of detail.

Threat models in general and Trike in particular are also very good as communication devices. We lay out all the threats to the system and their associ-

ated risks in a clear, easy to understand fashion, and provide stakeholders without security background a clear understanding of the security situation as it stands. While technical personnel will find detailed attack graphs useful in plotting remediations, even non-technical personnel can instantly appreciate and work with higher level threat data. Trike threat models also communicate where risks lie in a system, and can also be used to compare risks between systems. In addition to allowing actuarial decisions to be made about a system, this can guide purchasing decisions if outside systems are being evaluated, focus risk management or software improvement processes, and evaluate alternate design possibilities. Risk calculation can even feed back directly into the process for large systems, as an initial baseline set of threats can be generated and, based on risk data, different areas can be selected for full attack graph generation.

Not all sections of the Trike methodology are presented at the same level of confidence. As the methodology is currently under heavy development, some sections have not been fully tested against real systems and care should be exercised in their use. Many sections of the methodology are likely to change without notice, and new versions of the methodology will likely be incompatible with older ones. Specifically, supporting operations (especially), the state machine, and use flows are all experimental, as, obviously, are attacks generated from them. Attack graphs, while currently somewhat ad-hoc and currently under development, have been tested (as have the requirements model and threat generation). The risk model has been tested to a certain degree, but should only be used as a guide at this point in time; refinement of the model is planned. The current draft of the paper does not include examples. A final version will be forthcoming soon with a fully worked sample threat model.

1 Requirements Model

All threat models must begin with an understanding of what the system is intended to do. Trike looks at who interacts with the system, what things the system acts upon, and the actions taken by actors that the system is intended to support. We go further to look at what rules exist in the system to constrain those actions, and tie all of this information up in a convenient tabular format. Once the requirements model is complete, we can skip ahead to the threat generation (see section 3.1 on page 8), or continue on to the implementation model (see section 2 on page 5) and do both threat generation and the attack graph at once.

1.1 Actors

An actor is a human being who interacts with the system in some way. Actors must interact with some part of the system which is in scope for the threat model. For instance, the backup administrator is not an actor, unless you are analyzing the backup system. Likewise, the software developer is not an actor unless you are analyzing a source control system. In some rare cases, a piece of software might be an actor in the system, if, for instance, it is capable of autonomously taking actions against assets without prior instruction from a human. This case is very rare however, and you should think carefully if you think that you have a non-human actor — we have yet to encounter one. Simple scheduled jobs and the like definitely do not qualify for actor status. The actual entity being described by an actor is the role that the actor takes. Each set of permissions in the system should be a distinct named role. A human may have more than one role in the system and may change from one role to another while using the system (see section 2.1 on page 5 for more details).

1.2 Assets

Assets are normally discrete data entities, but sometimes physical objects, which feature in the business rules of the system. Assets are things which are inherently meaningful in the problem domain of the system, not merely in the way the system is implemented. For example, a widget that a company is shipping across the country (or the data that represents it in the system) is probably an asset, while the password of a blog user is not an asset. While the password may be very important in the way the system is implemented, if there was a way to strongly identify a user without a password or equivalent token, a password would not need to be part of the system at all. We care about the data that needs to be secured, not artifacts of how it is secured.

Assets are fairly specific, tangible pieces of data. For instance, the reputation of the developer who wrote a piece of software is not an asset to the system, nor is the uptime of the system. The system itself is an asset only if the system is self-referential. The system software might be an asset if the system installer is in scope for the audit, but otherwise would not be. Likewise, system configuration files are almost always not assets, nor are the machines the system runs on. There might be an exception to the first if the software provides an interface to edit its configuration, but even then, the data in the file would be the asset, not the file itself.

Assets can normally be associated with a dollar value to the client. While the generation of such values is an imprecise science at best, a company can normally come up with values for how much an average transaction costs them or what the productivity cost would be if a crucial service was unavailable. See section 4.1 on page 12 for more discussion on assigning values to assets and the trade-offs involved in this decision.

Identifying assets can be one of the more difficult parts of the requirements model, and if you are going to miss something, it is probably an asset. One place to start is by looking at what information each actor uses in their interactions with the system. Also, look at what nouns you use repeatedly when talking about the purpose of the system. What transient or persistent data is inherently meaningful to the system? Unique assets are almost always treated differently in the rules — if two or more assets are treated identically in the rules, you have an opportunity to simplify your model by combining them, and you will probably discover later that this is the correct choice. The greater the number of assets you model in a system, the more detail you can include in the rules. However, the number of threats increases geometrically with the number of assets.

1.3 Intended Actions

Actions are things which actors do to assets. Our primary focus with Trike is on threat modeling for software, and thus while the decomposition of actions for data assets is fairly fixed, the same decomposition for physical objects is very much a work in progress. That said, any (virtual) action can be decomposed into one of "create", "read", "update", and "delete" (CRUD), or a compound action based on one or more of these actions. Create, read, and delete actions are generally fairly obvious. An update action might alter an entire instance of an asset, or only some small part of one (see section 1.4 on the following page below for more information). Copy can be thought of as a combination of read and create, transformations of data can be considered to be instances of update or read and update on a single asset, move is a combination of read, update, and delete between a pair of assets, and a closed control loop can be seen as an instance of read and update, again between a pair of assets. This ability to break down arbitrary actions into combinations of these four atomic actions is at the core of the Trike methodology, as it is inherently related to how we generate threats later on.

Some circumstances do call for another more exotic action, but this is only very, very rarely needed, to the point that we hesitate to even include it. If you deal with a system which is specifically intended to move around code and execute that same code as part of the core function of the system (and not merely as an implementation-defined means to an end), then you may have an additional "invoke" action in your system. In our experience, this is rarely actually the case. No other actions are ever needed for virtual assets.

We make a distinction between actions generally and those actions which the system is intended to support. Each intended action is a triple of action, asset, and rules, with the rules for an action including those roles or actors which may perform it. Together, the intended actions form a complete formal description of what the system being analyzed is trying to achieve.

When constructing the list of intended actions, we look at only those actions which are supposed to happen during the normal use of the system as designed, regardless of whether or not the system as implemented allows them. Unintentional behaviour, be it an easter egg or a vulnerability, is not included. Every intended behaviour of the system must be expressed in the intended actions. This means that every actor and asset must appear at least once. Some actions may not have an actor, and some may not specify any specific requirements for the actor, as although actors are called out separately they are simply another kind of rule applied to an action. In-

tended actions without actors denote automatic jobs or other non-human agents acting on the system.

To identify intended actions, examine each asset and actor pair in turn, and consider each of the four action types. Is this action supposed to occur within the problem space the system is intended to solve? Under what circumstances should it happen? Given these answers, the set of actions should be obvious, as should the beginning of the rules for the action. It is important to note that there are a number of things which may be critical in the operation of a system, especially from a security perspective which are not, in fact, intended actions, as they exist only in the implementation of the system, not in its requirements directly. These actions will almost always change the state of the actor taking them or of the system without directly modifying any assets in the system. Events such as logging into the system fall in this category, and are covered in section 2.1.

1.4 Rules

The complement of the intended actions are the rules that apply to each action. Rules define in which circumstances an action can occur. The rules for an action are a set of declarative sentence fragments, connected by logical connectives (and, or, and not). Rules should be consistent in wording and terminology, for ease of analysis. Rules can be stated in a positive or negative form, the negative being the contrapositive form of the entire tree. One of the most basic rules that almost all actions will have is "actor is in <role>", where the role is one of the named categories of actors. Other categories for rules include the frequency that actions can be taken or when they may or must occur, what portions of an asset can be affected by an action, specific properties of the actor or asset, what data must be recoverable about an action after the fact (simply stating that an action must be logged is not sufficient to capture the intent of logging), other actions that must be taken before or after the action in question, and relationships between instances of different assets or actors to assets.

1.5 Actor-Asset-Action Matrix

The actor-asset-action matrix is a convenient form to represent all or almost all of the data about the requirements model in a grid format. The columns of the matrix represent the assets in the system, and the rows represent the roles that actors can take on. Each cell of the matrix is again subdivided into four, for each of the actions of CRUD. When using the tool, each action-cell of the matrix can be set to three different values, to indicate an allowed action, a disallowed action, or an action with rules, and a list of slots to add rule trees is generated. Without the tool, a mark should be placed in each cell of the matrix and the rules trees written up separately.

2 Implementation Model

Once the requirements have been formally defined, information must be gathered about the implementation. With an understanding of what the system is intended to do, this can be ordered much more easily. We start by looking at those actions in the system which do not fit into the intended actions framework and how actions interact with the state of the system. We then look at how the different software and hardware components of the system as implemented fit together in the data flow diagram. Then we map from the actions and state of the system into the data flow diagram. From here, we can proceed to the threat model (see section 3 on page 8) and attack generation.

2.1 Intended Actions vs. Supporting Operations and the State Machine

Every actual system has steps which the user takes in the system which are not included in the set of intended actions. These operations are how the user interacts with those business rules of the system which affect how and when they may take actions and what requirements they must satisfy to do so. In other words, these supporting operations affect the state of the application itself, as opposed to the assets. Login is a prime example of such a supporting operation — it places the user in a new role, which is a prerequisite for other, intended actions within the system. Determining the set of supporting operations is similar to determining the set of intended actions.

In order to find a set of possible operations, we must first find the set of possible objects of those actions.

In one sense, supporting actions only have a single object, the state of the system. However, for our purposes we need to approach the subject at a finer granularity. Construct a table, and for each state-like requirement in the rules for intended actions, select a state tag which consists of an identifier and the type of object in the system which it applies to. The "type of object" should be an asset in the system, an actor, a specific role, or something else similar. Thus, one could have a "posts available" state identifier which could apply to assets of type "blog", or a "logged in as user" tag which could apply to actors, denoting the actor being in the role "user". Separately, one could have an identifier "account disabled" which would apply to a role "user" (not, obviously, to the actor taking on that role). Some subset of the state tags will be starting states for the system, and this should be noted in the state table.

Once the set of state tags is determined, we look at the set of specified supporting operations in the system and formalise them as a set of state addition and subtraction operations, and potentially a set of rules for each operation. Note that in some cases, especially with complex systems, this process may be recursive to a certain degree, as new state tags will need to be generated based on the rules for supporting operations. Some supporting operations may not be performed directly by an actor. Some of the intended actions may also have side effects which affect the state of the system, especially if the actions have rules limiting the frequency at which they may be taken.

Once the set of state tags and supporting operations is complete and their relationship to the intended actions noted, we have a state machine for all acts possible within the system. Portions of this state machine may be drawn out, but for non-trivial systems this diagram will often prove prohibitively large or not easily diagrammable at all. It is important to check that all states in the system are actually reachable from the starting states.

Supporting operations specifically and the state machine in general are currently highly experimental features of the Trike methodology and are likely to change. Care should be taken when using these features in a production audit; while we believe that are likely to be no errors introduced in the model from them, they may prove to be cumbersome or problematic, especially in large systems. There is currently no tool support for any of these features. It will be implemented in an upcoming release, but the features themselves may have changed.

2.2 Data Flow Diagrams

The next portion of our model of the system's implementation comes from a data flow diagram. While there are multiple diagramming techniques which could potentially represent this information, this method seems to be the best fit for our needs. A DFD gives a logical depiction of the implementation of the system and shows the large-scale architecture of the system. It shows what entities exist in the implementation of the system, and along what paths these entities exchange information.

The entities in a DFD consist of processes, data stores, external interactors, and data flows. A process is a part of the system which does something, such as manipulating data, taking actions in the problem domain (intended actions), changing or verifying system state, etc. Processes represent the software (or collections of software, etc.) which carry out these actions, and are recursively composed in the diagram. Each DFD, except for the top level context DFD, is the expansion of a process in the next higher level DFD. A data store is a resource whose primary function is to store data for later retrieval. Note that many databases qualify as both a process and a data store. An external interactor is a process which is out of scope. For many purposes, actors are also considered external interactors. A data flow is a path from one DFD entity to another, along which specific data moves. Processes may talk to each other or to data stores or external interactors, but data stores cannot talk to each other directly (there is no agency to perform the connection without a process). Each data flow in or out of a process to be expanded must appear in the new lower level DFD. Any given DFD should only show those data flows which connect to processes in this DFD — flows between external interactors are normally not included, although if human processes are part of the DFD, then the relevant flows between actors should be included.

The top level or context includes every actor and every out of scope system or process utilized by the system. As lower level DFDs are expanded, trust

boundaries in the system should be noted, be they boundaries between networks enforced by firewalls, boundaries between machines, process boundaries, or more literal trust regions enforced by authentication technologies, either those part of the system or underlying it. For our purposes, DFDs must be decomposed until there are no longer any processes which cross trust boundaries.

Once the basic DFD has been constructed, markup must be added to it to capture more information about the implementation. At the most basic level, we need to capture the technologies used in each element. For processes, this means the host OS, libraries, platforms, or underlying applications, and all relevant version information. For data stores, we need to know the type of data store, be it a file store, a database, or a registry entry, and again, relevant version and host information. For data flows, we need to know about network protocols at least, and ideally which applications on each end are the actual data flow endpoints, if this information is not captured directly in lower level DFDs. Data flows which cross trust interfaces are especially interesting, and the end points for such flows should always be well documented. Unsurprisingly, we also want information about trust boundaries and what enforces them, including as much detail as possible. Where security technologies are in use throughout the system, they should be specifically called out, including encryption, authentication, and authorization systems, firewalls, certificates, passwords, etc.

For large systems, the correspondence between the physical or network layout of the system and the application level DFD can be complex. In these cases, a network diagram of the system is an essential supplement to the DFD. The DFD should include at least one process which runs on each machine in the network diagram, unless they are specifically not included in the system (and in this case, their inclusion in the network should be considered carefully, as they represent a latent risk to the security of the system). Lower level diagrams provide a number of useful functions. They allow you to sanity check the DFD, ensuring that all the systems and (especially) all the trust boundaries really are the way you think they are. They let you unload complex details from busy DFDs. Network diagrams especially are good for this — you can list all of your network level trust boundary technologies on the network diagram only, and do the same for network technology details, and only the actual trust boundaries along with the application level details on the DFDs themselves. Network diagrams or other lower level diagrams are also great for pinpointing areas where trust boundary hopping can occur, such as situations where two data flows which are called out as being on completely separate networks are actually only on separate VLANs on the same switch. More than two sets of diagrams may be needed for complex situations, e.g. an application level DFD, a TCP level DFD, and a hardware level DFD, especially in situations like the previous involving VLANs.

Trike does not yet have tool support for DFDs, and they can instead be created in most structured drawing tools. DFDs will be implemented in an upcoming release.

2.3 Use Flows

Once we have a full set of all actions intended and supporting, the corresponding state machine, and a clear understanding of the architecture of the system, we map from state machine to the DFD and come up with all the use flows for the system. Use flows are used to map actions in the implementation and show all the ways that assets and application state can be affected from inside the system. They provide the basis for understanding complex attacks within the system.

To construct use flows, we take each intended or supporting action in the system and trace the paths through the DFD that will implement that action (there may be more than one way to do it). We break the use flow into segments each time it passes through an external interactor, including users. Each use flow will normally have a set of preconditions and postconditions in the state machine. Actions which are broken into multiple segments by returning control to a user may need to have additional state transformations added to reflect enforced work flows within the system. For example, if posting a blog entry is a two stage process consisting of submitting the initial post and then reviewing and approving it as displayed back by the system, a state addition should added to the first use flow segment with a state tag of “post submitted”, and the second use flow segment should

require this state for it to be entered (with the tag being deleted at the end of the second use flow segment).

If alternate legitimate paths through a use flow exist, the flow should be forked at the point at which control flow changes (regardless if it is passing through the same nodes or not). This can be used to handle failure responses, etc. If the user's actions can affect which nodes the path goes through, this should also be shown as a branch in the use flow (for instance, if the user can select which database to store information in).

Once the basic set of use flows is complete, markup must be added here as well. The markup added to the DFD should not be duplicated here. Rather, we want to mark where elements of the state machine are implemented in the DFD. We mark the moment (or location, if you will) where each intended action is complete, the location where each precondition or other rule is enforced, and the moment when the state changes occur. Important side effects such as logging should be noted, as should the content of the messages traversing each data flow in the use flow.

Once the use flows are constructed, some basic sanity checks should be made. Are there any elements in the DFD which are in scope but are not touched by any use flow? These imply that either these DFD elements are unnecessary and their presence in the system should be investigated, or that some number of actors, assets, or intended and supporting actions is missing. Also, any time there is more than one use flow for an action or more than one way to enter a state, all relevant flows should be checked to ensure that they enforce the same set of constraints.

Use flows are currently a highly experimental feature of the Trike methodology and may change. Care should be taken when using them in a production audit; while we believe that there are likely to be no errors introduced in the model from them, they may prove to be cumbersome or problematic, especially in large systems. There is currently no tool support for use flows; the same ad hoc tools used for DFDs can be used, although they will be somewhat cumbersome. Use flow support will be integrated into a future release of Trike.

3 Threat Model

Once we have a full model for both the requirements of the application and the implementation of the application, we have all information needed to start on the threat model proper. The first part of the threat model can be started with only the requirements model and the implementation model will not change these results. Once we have the set of threats for the application and the implementation model, we proceed to building the attack graph and examining the actual (as opposed to specified) system to verify all weaknesses in the system. This done, we can determine the vulnerabilities to the system and apply mitigations. Of course, not all of these steps must be done for all areas of the system. If risk calculations are performed once threats have been generated, specific components can be singled out for attack graph generation, etc. This does, of course, risk missing some attacks which cross system boundaries, and whenever possible a full threat model should be performed.

3.1 Threat Generation

Threats describe what, specifically, could go wrong with a system in the application domain. They are the negative eventualities that we are attempting to prevent. In terms of the requirements model, threats are anything which is more or less than the intended actions, as these represent everything that can be done in a secure system. Threats are never technology specific or implementation specific (those are attacks) — instead, they are couched directly in terms of the business rules of the system. Threats are also always events, not actors who might be a risk (organized crime would be an attacker which might attempt to realize a threat, not the threat itself). The set of threats against a system is purely deterministic, given the actor-asset-action matrix (and associated rules). Each threat will later serve as the root of an attack tree within the larger attack graph.

We have a fairly simple taxonomy of threats. All threats fall into one of two categories, either denial of service or elevation of privilege. A denial of service threat occurs when an actor is prevented from taking a legitimate, intended action in the system when acting in accordance with the rules for that action. An

elevation of privilege threat occurs in one of three situations: When an actor performs an action which no actor is intended to perform on an asset (an entirely disallowed action), when an actor performs an action on an asset despite the rules for that action (specifically disallowed action), or when an actor uses the system to perform an action on some other system's asset. This last instance is the "social responsibility" threat, and represents such things as open mail relays which can be abused by spammers, etc. While this may not directly affect the security of the single system in question, it may lead to reliability problems (if nothing else), which will certainly cause real issues.

Spoofing, sometimes considered a threat in other methodologies, is actually an attack in most cases. In some specific cases (such as a cryptographic system specifically intended to prevent it), it can be a threat, but is just an instance of an elevation of privilege threat where an actor is able to violate a rule specified for the action in question stating that spoofing must not be possible. Tampering with data and information disclosure are likewise just instances of elevation of privilege.

Generating threats is quite simple, given the data we have. One denial of service threat is generated for each intended action. Then we invert the set of intended actions and take the set of disallowed actions, generating an elevation of privilege threat for each one, the set of entirely disallowed actions. Next, we take each intended action with further rules and generate an elevation of privilege threat for it, the specifically disallowed actions (actions performed outside their rules). Finally, we add what we call the social responsibility threat, namely the threat of an actor using this system to take an action against another system. All possible threats are thus encompassed within this schema.

Even without progressing further into the threat model, a set of threats can be quite useful. We'll see in section 4.5 on page 13 that a risk analysis based on only the threats has value; however, even without that, we can do basic requirements analysis and simply look at the number of threats against a system as a guideline for dealing with the complexity of the security implications of a proposed system. Also, the set of threats can be very useful when moving from the requirements phase to the design phase, as it provides a canonical list of potential security issues to design around.

3.2 Attacks, Attack Trees, and the Attack Graph

Given an implementation model and a set of threats, we can begin to look at ways in which those threats may be realized. An attack is a threat-specific, implementation-specific, or technology-specific step an attacker could take to realize or help to realize a threat. Attacks are organized into attack trees, which are hierarchical descriptions of how an attacker could realize a specific threat to the system, using this implementation. The attack tree is made up of tasks and subtasks. The root node of each tree is a threat, and the children of each node describe in increasing detail how an attacker could accomplish the task in the parent node. Other than the threat root node, all the nodes in an attack tree are attacks. Each node's children are subgoals for the node, and together, the node's children should specify every way that this node could occur. In addition to goal type nodes, attack graphs can contain logical connectives. Some nodes may require all of their children to be accomplished in order to be accomplished themselves, while others may require only a single node to be accomplished.

Normally, a complete attack tree will not need to be generated for every threat. Rather, you only need to expand an attack tree until there is enough information to reasonably decide whether the risk caused by the threat has been reduced to an acceptable risk level. If it has not, then you need to continue to generate a complete attack tree for the threat, so the actual weaknesses can be understood and mitigations can be implemented. A leaf node of a complete attack tree will locate a specific type of attack in the data flow diagram and source code. You cannot expand such a leaf node any further, hence the name "complete attack tree". Leaf nodes can have weaknesses, vulnerabilities, and mitigations attached to them, as described below.

It may seem somewhat odd to concern ourselves so heavily with the means of attack against an application when we are interested in threat modeling from a defensive perspective. The critical issue for a defensive perspective, however, is to ensure that we find all

weaknesses, not just a sufficient number to compromise the system. Only by enumerating all possible ways to attack the system, once we have determined what we are trying to defend, can we ensure that we are adequately defended against all attacks.

A single attack may be used in realizing multiple threats. Thus, while each individual threat has an attack tree specific to that threat, there is a larger directed attack graph encompassing all attacks against the system. All attacks help to realize at least one threat.

The generation of the attack graph is not currently a fully automateable process. As we are not attempting to prove the correctness of the software formally, the experience and knowledge of the auditor working on the threat model must be relied on at certain points, and attack generation is one of those points. High level attack generation follows a straightforward attack tree skeleton, and thus generation proceeds programmatically to a certain point. Beyond that point, the auditor relies on their skill and the library of attacks to finish the tree to the desired depth. Developing a lower level attack taxonomy on an ad-hoc basis may prove to be a very useful endeavor for an organization using threat modeling heavily. We are hoping to develop a more canonical and carefully considered attack taxonomy in the future.

The quality of attack generation is limited by the accuracy of the system description and the quality of the attack library used in generation. If the set of intended actions and (especially) the rules for those actions is not complete and correct, the correct set of attacks cannot be generated. The same applies for the supporting operations, the state machine, data flow diagram, and use flows, although there is more flexibility in this second set. The more data is captured there, however, the more accurate the attack trees will be.

Attacks, attack trees, the attack graph, weaknesses, vulnerabilities, and mitigations are fairly solid features of Trike currently. We are expecting that they will become more refined and the way they interact with other parts of the model may even change considerably, but they are well tested as they stand. Currently we do not have a released tool that supports working with attacks, but we are currently working on developing one.

3.3 Weaknesses

A weakness is a problem in the system which allows a leaf node in an attack tree to work, regardless of whether the attacker can actually realize a threat using it. Any flaw or error with security implications in the source code or application or host configuration of a system can be a weakness. Something can even be a weakness without allowing for an attack to succeed — if an organization has decided that certain constructions have too much risk, they can be considered weaknesses, even if nothing exploitable exists. In general, though, a weakness is the reason that a specific attack succeeds. A weakness is the opposite of a mitigation.

Weaknesses are the most direct point of contact between a code audit and a threat model. Although the threat model should be guiding the focus of the code audit and indeed the entire security process, bugs found in a code audit are equivalent to weaknesses in the threat model. Starting with either a complete or mostly complete attack tree, an auditor can perform a targeted sweep of the code, picking out weaknesses. Likewise, if any security issues are found in the code which do not correspond to attacks in the attack graph, the graph should be examined to determine where it is incomplete. The results of changing the graph should be propagated back into the code audit to ensure that there are not areas that were previously missed.

3.4 Vulnerabilities

A vulnerability is an unmitigated path through an attack tree from one or more leaves to the threat, wherein all conditions for each attack are met. This specifies a weakness or a collection of weaknesses which allows an attacker to implement a threat. Vulnerabilities are by definition exploitable, and are created automatically given accurate and complete attack trees. A path through an attack tree is considered a vulnerability if there are no mitigated nodes in the path, including all required branches. Vulnerabilities are used heavily in the risk model (see section 4.3 and 4.5 on page 13 for more information).

3.5 Mitigations

A mitigation is a safeguard that reduces or eliminates the risk associated with a particular weakness. The goal of mitigation is to reduce risk to an acceptable level, as defined by the stakeholders. A mitigation can either reduce the likelihood of a successful attack, or reduce the impact of a successful attack. Mitigations must be considered carefully. The best thing to do is to apply an iterative process and perform the same steps of threat modeling and code auditing against mitigations that you would against any other system, as a mitigation can be attacked in the same way as any other part of the system. Mitigations can occur at any level, from the addition of new rules or the alteration of existing rules to code level bug fixes. Adding new technologies at the DFD level can be a relatively easy fix for some architectural problems, but it is very important to look over these kinds of fixes carefully, as it can be easy to add new issues without actually fixing the existing ones. Furthermore, some things do not actually mitigate against any threats. Logging or audit trails are definitely a good idea from a best-practices standpoint, but they don't actually mitigate against any threats directly. If there are rules requiring the non-repudiation of actions, logs will be useful, but only as far as they are correct and trustworthy.

3.6 Attack Libraries

Attack libraries are one of the more useful time saving features of the Trike methodology. Attack graph generation, especially if done in-depth, can be very time consuming. However, there's no reason to start from scratch each time. Certain patterns appear again and again in attacking systems. There are only so many ways to circumvent a login requirement in an application, for instance, so there are high level patterns there, mirroring the yet higher-level patterns embodied in the threats themselves. Then, there are the lower-level patterns. Each category of weakness has specific ways in which it can be exploited — all cross site scripting exploits share certain similarities. Furthermore, once you've attacked a certain technology in certain ways, you'll find that these too, repeat. A library of attacks can thus be built up relatively easily, with enough experience.

Attack libraries are where the true power of the technology annotations on data flow diagram and use flows comes into play. By mapping between the kind of attack that is being attempted and the technologies used by relevant DFD elements, the appropriate portion of the attack library can be selected automatically. In many cases it may be possible for the subtrees from the attack library to simply be referenced as-is in the system-specific attack graph. However, in many cases, they will need to be copied into the graph and customized for the specific system in question. Either way, this makes the generation of in depth attack graphs much more rapid. Furthermore, the more audits an organization has done, the better their library of attack subtrees will become.

Trike will eventually include functionality for managing attack libraries in an intelligent fashion, but this is not currently a very high priority, as it is an operational as opposed to methodological concern. Trike will also not be shipping with more than a small demonstration attack library. Attack libraries will require a certain amount of maintenance work to keep up with exploit releases for technologies, new software versions, etc., and as such it's not something we will have time to maintain.

4 Risk Model

While risk modeling and risk management is a core feature of the Trike methodology, this model is still highly experimental and will change. We are attempting to reach first an operational risk model and then an actuarial one, but we do not consider this model to necessarily be ready for use in either role. That said, even a rough estimation of risk allows us an incredible amount of leverage within our framework, and that level of certainty is easily achieved.

One of the key concepts within the Trike risk model is that of in and out of scope risks. When examining the potential risks to an application, it is important to understand exactly which parts of the system and which risk factors to those parts are being modeled. For instance, it is quite reasonable for a group to simply decide that attacks against the cryptography of a properly implemented version of the AES algorithm are out of scope for the purposes of their threat model. They are assuming any risk there and

declaring it to be both acceptable to them and out of scope for the model. On the other hand, it's equally reasonable that such a risk would be explicitly considered in scope in some high-risk situations. While we would not necessarily recommend doing so, a team could also decide that they're confident in the platform that they develop applications on top of, and will thus assume the risk for all attacks which might succeed via the host operating system, runtime, etc. It is very important to extend the threat model, especially the attack graphs, to include all of these things — the model itself should never be pruned, as this sort of a priori pruning will result in large swathes of vulnerabilities being missed entirely. Risks can be assumed, changing the prioritization of weaknesses and vulnerabilities. In the following discussion, we will assume that risks are being scoped appropriately in all cases.

4.1 Asset Values, Role Risks, Asset-Action Risks, and Threat Exposures

As with everything in Trike, we start at a high level when we consider risk. Each of the entities in the requirements model has associated data relevant to risk calculations. We start with assets and give each asset a dollar value, based on its inherent business value to the organization. This value should be decided by the business, not the auditor, and represents a rough estimate which is nonetheless useful as a relative value within the system. Next, we look at the actions that can be taken on an asset and choose relative values for their undesirability, from one to five where five is most undesirable. Each pair must be ranked twice, first with one value for when this authorized action cannot be completed in accordance with the rules, and then with a second value for when an attacker completes this action despite the business rules which disallow it. Obviously, unintended actions do not need to receive a value of the first type, and actions which have no associated business rules (e.g. an anonymous user should always be able to do this) do not need to receive a number of the second type. Last, we look at actors, and give each actor a risk level, again between one and five, with highly trusted actors being a one and untrusted (likely anonymous) actors being a five.

With this data defined, we can now give an exposure value for each threat we've generated. We define exposure as the value of an asset multiplied by the action-specific risk. With this value, we can rank our threats in order of their seriousness to the organization.

It should be noted that although the resulting exposure is technically denominated in dollars, this does not represent a real dollar value risk to the organization. Instead, it represents a ranking of threats based on the values of the assets they pertain to. A further note is that this asset value model is currently somewhat intentionally naive. An asset does not have a single value to a company, and although with action specific risks, we capture some of this information, reality is much more complicated, of course. Specifically, it does not sufficiently take into account the differences in the value of an asset based on who can take disallowed actions, and it does not take into account the asymmetric nature of asset valuation. For example, in a supply chain system, an instance of an asset of customer order information might be worth the average dollar value of the order to the organization filling the order, but could be worth much more to the customer, who has business continuity and competitive intelligence issues if there is a security breach and a threat is realized against that asset.

4.2 Weakness Probabilities and Mitigations

Once we have determined the set of weaknesses in a system, we can find specific probabilities of exploitation for them, too. First, we rank each weakness on thee scales, each from one to five. The first scale is reproducibility, which looks at how easy a given weakness is to reproduce. A complicated race condition which requires a very specific system state might be a one, if there was no straightforward way to produce that state, while an unauthenticated cross site script would be a five. The second scale is exploitability, where we look at how technically easy an exploit is to perform. A simple canned cross site script would again be a five, while a blind heap injection would be a one. It is important to be aware of security trends when assigning these numbers, as they can change; for instance, if a canned exploit for the previously mentioned heap injection comes out, exploitability

suddenly got much easier. It may pay to be somewhat conservative here, especially when dealing with widely used software. The third scale denotes the risk value attached to the least trusted actor able to technically effect this weakness.

To calculate the probability of a specific weakness being exploited, we multiply the reproducibility, the exploitability, and the actor risk value. This probability factor, although not quantitative by any stretch of the imagination, provides a somewhat coarse but still very useful ranking of all the weaknesses in the system. Note that exploiting a weakness does not imply fully implementing a threat; rather, it merely implies exploiting the local weakness in the system. The probability of a weakness being exploited is a purely technical measure. Once we have examined vulnerability risks, we can return to weaknesses and rank them taking into account business impact.

A mitigation will rarely remove a weakness entirely; more often, the bar for the attack is simply raised to a level where the organization responsible feels comfortable assuming the risk of the associated vulnerabilities. Thus, the primary effect of mitigations is to reduce the reproducibility or exploitability of a given weakness or class of weaknesses, or to restrict access to a more trusted class of user.

4.3 Vulnerability Probabilities and Exposures

Once we have ranked weaknesses, we can extend that information to the vulnerabilities which they embody. For each vulnerability, we define a probability by looking at the graph of weaknesses which implement it. The set of vulnerability probabilities and exposures is not directly useful to us, but it allows us to push weakness probability information up to the threat level and threat exposure information down to the weakness level.

If there is only one way to exploit the vulnerability, the probability of exploit is the lowest probability of the set of required weaknesses. If there are multiple possible ways to implement the vulnerability, the probability is the highest of the probabilities of each sufficient subset of the set of applicable weaknesses. Another way to understand this is that at each "and" node in the attack tree, we select the least probable sub-node, and at each "or" node, we select the most probable. This represents the situation where an attacker can pick the path through the attack tree which is easiest for them to accomplish ("or" nodes), but is constrained by the hardest obstacles on any given path ("and" nodes). We define the exposure for a vulnerability as the sum of the exposures of the threats this vulnerability makes possible. Again, this is not a quantitative figure but a useful set of guidelines.

4.4 Threat Risks

Once we understand both the exposure for each threat and the probabilities associated with the vulnerabilities that implement that threat, we can calculate a threat risk value. For this, we simply multiply threat exposure by the largest applicable vulnerability risk. This provides us with a set of values which take into account the technical security issues and relate them to the business impact of those issues.

4.5 Using the Risk Model

A fully-worked risk model, even a coarse one like Trike's current system, is very powerful. The sets of threat exposures, weakness probabilities, and threat risks build on each other, and together they provide a very useful set of capabilities. The first set, threat exposures, provides an immediate and high level overview of the risks in the system, without requiring in depth analysis. This allows later work to be focused on those areas of the system which are most important. It also provides a good focus for emergency response planning and incident response. Although later portions of the risk model provide more information, they are also colored by assumptions about the system based on the implementation. In an emergency, an understanding of the most basic prioritization of business importance can be critical.

The set of weakness probabilities is most useful from a remediation standpoint. Weakness probabilities directly provide an ordering of all potential places for mitigations in the system. This allows the threat model to integrate closely with the overall software engineering effort for the system in question. As the vulnerability probabilities embody a dependency analysis of the weaknesses in the system, we can perform an attack graph traversal and provide an order-

ing of weaknesses based on which ones contribute the greatest business exposure to the system. First, we determine the set of weaknesses which are constraining the probability for each vulnerability, and then we simply sum the exposure values for each vulnerability that each member of the set of constraining weaknesses contributes to. This value provides a direct ranking by business impact, and tells the organization explicitly which weakness to fix first. This analysis is necessarily iterative, as after each mitigation is in place, the rankings must be recalculated, as new weaknesses may have entered the set of constraining weaknesses.

Threat risks are perhaps the most useful, as they are a living model of the risk in the system. They provide an interactive view of risk to the system as assumptions change. For instance, if someone announces a tool which automates a class of exploit to which the system in question is vulnerable, the operations staff for the system can instantly see what the business level impact is, and respond accordingly. In addition, recomputing threat risk while examining different potential mitigation strategies allows a further level of decision making, making it possible to find those weaknesses which will have the largest impact on the greatest number of separate threats if mitigated. A wide variety of speculative risk analysis can be done at this level, not just operational ones. For instance, if multiple designs have been threat modeled, threat risks are an appropriate level at which to compare the security impacts of design decisions, and indeed the only possible level at which this can be done.

5 Work Flow Notes

Although there is a definite hierarchy to the information in the threat model, Trike itself does not require any specific work flow, and can be adapted to almost any situation. Obviously, you will not be able to generate attacks until you have threats, but you could, if you desired, start cataloging some obvious technology-related weaknesses before examining the overall requirements of the application. We don't recommend working in this way, however, because in our experience the level of understanding generated in examining the system requirements will allow you to work much more quickly when work begins on actual code auditing. A brief overview of two sample work flows may be useful for users attempting to implement Trike in their organizations, and thus follows.

5.1 Full Software Life Cycle Work Flow

In the ideal situation, threat modeling should be a formal part of the software development process from the very earliest stages of development. In this situation, the documentation required for the threat modeling process should be integrated into the primary system documentation, and in the ideal case, the threat model should simply be integrated as a core component of documentation. While the model below obviously maps most directly to a simple waterfall development model, these same steps occur in all development models. Trike strongly supports information flowing back up the chain and adapts easily to piecemeal expansion, and as such fits just as easily into spiral development or XP/agile models.

Requirements

The requirements document for the application should be broken out into the assets the system will act upon, the actors who will take those actions, and the intended actions which will be supported. A carefully analysed set of business rules should be specified at this stage, which will obviously be useful not only for the threat model but also for later testing purposes. Threats, when generated, should be explicitly included in the security section of the requirements document. Security risk analysis should also be included, along with more traditional scheduling and performance analysis. The most relevant project personnel to work with at this phase will be business level people and project managers.

Specification

The specification document for the application should include a full DFD for the application, a listing of all intended actions and supporting actions and the state model required to support those actions, and use flows for all actions. The specification should also detail how and where all the business rules from the requirements document will be enforced. Attacks

against the system should be generated, and the security section of the specification should include both the attacks and how the attacks will be mitigated. Risk analysis should be performed here to ensure that appropriate responses to attacks are developed and that implementation efforts are spent correctly. If problems come up with the requirements when the application is designed, the requirements document, including the generated threats, must be updated, and this information must be propagated forward to ensure the correctness of the threat model. The most relevant personnel to work with at this phase will be architects and lead developers.

Implementation and Testing

As code is written, it should be checked against the specification to ensure that all the necessary mitigations are built, and that the business rules are implemented as specified. Test suites should be used whenever possible to automatically confirm the enforcement of the business rules and the correctness of mitigations. If new attack vectors are found during the course of implementation or the code as written deviates from the specification, the specification document must be updated to ensure the correctness of the threat model. The most relevant people to work with at this phase will be development and test leads, with operations leads stepping in for configuration and network issues.

Audit

When the code is audited, the process should be very straightforward. A full audit is still necessary, to ensure that nothing was missed in the development of the attack graph, but this will primarily consist of double checking all potential weaknesses and mitigations, along with reading the code. Any errors here should propagate back to the specification as they are fixed. Development, test, and operations leads and architects should all be involved here as needed.

Training

Once the audit is complete, the categories of errors found during the audit should be evaluated, and training scheduled for the relevant personnel to reinforce skills in problematic areas. This can occur earlier in the process as well, and ideally should. If code is being reviewed as it goes into the source repository, there's no reason to wait until after the audit is finished to fix things. Training should occur at all levels, based on the issues found and the personnel involved in those issues.

Operations

In operation, the threat model should be used to direct patching efforts as new system vulnerabilities or attack vectors come to light. Also, the threat model can provide direction when allocating resources for future attack response by highlighting high risk paths through the system or high value assets. Operations personnel will be the primary interactors here, although business level people may need to be involved for risk assumption and future development decisions.

5.2 Pre-Release or Production Code Audit Work Flow

In many cases, full consideration to security is not or has not been given during the development process, and an auditing team is called in to examine code that is finished and either already in production or about to be released. While the full software life cycle process still calls for a final pre-production audit, there is far less work to be done in that case, as all the needed documentation already exists. A threat modeling driven audit should try to simulate the full life cycle to a certain degree, although the lack of information will necessarily result in more corrections to the model as new information becomes available. A threat model will start by working through any requirements information the team has available to them to extract the basic actor-asset-action matrix, along with attached rule information and the risk and value metrics. Although much of this work can be done in parallel, starting with the roles in the system is often helpful. Next, proceed to specifications documents for data flow diagrams and state models. Chances are that at this point, the team will need to start looking at the implementation directly for guidance and will need to generate these documents as they go. The process will thus be very iterative, because as the model is built, new attacks will go into the attack graph and suggest different areas of the

implementation to concentrate analysis on. The end result should be the same, however, because once a complete understanding of how the application is implemented is reached and the attack graph is fully formed, all weaknesses should have been examined. Once such an audit is complete, the documentation should stay with the project and be integrated into the next round of development for the project, so that the more efficient full life cycle model can be used.

A Glossary

Action Something an ACTOR does to an ASSET.

Actor A human being who interacts with some part of the SYSTEM.

Asset Data, or occasionally a physical object, which is featured in the business RULES of the SYSTEM.

Attack A task which, if it worked, would help accomplish a THREAT.

Attack Graph The set of all of the interconnected ATTACK TREES for a SYSTEM.

Attack Tree A tree of ATTACKS, rooted by a THREAT, comprised of all the ways that the THREAT can be realized.

Data Flow In a DFD, a link between two PROCESSES or a PROCESS and a DATA STORE.

Data Flow Diagram (DFD) Describes the PROCESSES, DATA STORES, and DATA FLOW in the SYSTEM. As used in Trike, it should include a full complement of technology annotations.

Data Store In a DFD, any location where data is persisted in the SYSTEM.

External Interactor In a DFD, a PROCESS which is either outside the scope of the SYSTEM or an ACTOR.

Intended Actions The ACTIONS the SYSTEM is supposed to make possible on each ASSET, taking into account the RULES.

Leaf Node ATTACKS at the edge of an ATTACK TREE which locate a specific type of attack in the DATA FLOW DIAGRAM and source code.

Machine Boundary In a DFD, the extent of a physical or virtual machine on which data is stored or processes execute.

Mitigation Something which prevents an ATTACK from realizing a THREAT.

Process In a DFD, any location where work is done on data in the SYSTEM.

Rule The circumstances under which an ACTION should be possible.

System The entire application, as defined by the scope of the threat model or audit.

Threat Something which shouldn't happen, i.e. something more or less than the INTENDED ACTIONS. A potential occurrence, malicious or otherwise, which might damage or compromise ASSETS.

Trust Boundary In a DFD, a TRUST BOUNDARY encloses a region where all actions occur at the same level of privilege, such as inside a single process. DFDs in Trike are expanded until no PROCESS or DATA STORE contains a TRUST BOUNDARY.

Vulnerability An unmitigated path from the leaves of an ATTACK tree to the THREAT.

Weakness The reason a specific ATTACK succeeds. A security issue in the SYSTEM, whether or not it allows a THREAT to be realized. WEAKNESSES are attached to the LEAF NODES of ATTACK TREES.

B References

While Trike itself is original work, a number of books contributed to our thinking, or document other threat modeling methodologies.

1. Kovitz, Benjamin L. Practical Software Requirements: A Manual of Content and Style. Greenwich, CT: Manning Publication Company, 1999, 1-884777-59-7.

2. Amoroso, Edward. Fundamentals of Computer Security Technology. Upper Saddle River, NJ: Prentice Hall PTR, 1994, 0-13-108929-3

3. Howard, Michael, and David LeBlanc. Writing Secure Code, 2nd Edition. Redmond, WA: Microsoft Press, 2003, 0-7356-1722-8.

4. McGraw, Gary, and John Viega. Building Secure Software: How to Avoid Security Problems the Right Way. San Francisco: Addison-Wesley, 2002, 0-201-72152-X.

5. Swiderski, Frank and Window Snyder. Threat Modeling. Redmond, WA: Microsoft Press, 2004, 0-7356-1991-3

6. Alberts, Christopher J. and Audrey J. Dorofee. OCTAVE^{SM} Criteria, Version 2.0. Pittsburgh, PA: Carnegie Mellon Software Engineering Institute, 2001, `http://www.cert.org/archive/pdf/01tr016.pdf`.

7. Common Criteria Development Board. Common Criteria for Information Technology Security Evaluatio 2005, `http://www.commoncriteriaportal.org/public/expert/index.php?menu=3`.