Threat Modeling for Secure Embedded Software

Dr. Gaurav Kumar Jain

Professor, Regional College For Education, Research and Technology

Email: gaurav.rinkujain.jain@gmail.com

Mr. Dipesh Bhardwaj

Developer, Regional College For Education, Research and Technology

Email : sft.er.dipesh3bhardwaj@gmail.com

Abstract— **As embedded software becomes more ubiquitous and connected – powering everything from home appliances and cars to aircraft and mission-critical systems – organizations must take additional steps to ensure that the code produced is both secure and reliable. Embedded software, however, presents a unique set of challenges for application development and engineering teams. To combat embedded software threats, teams are turning to strategies such as threat modeling, static analysis and penetration testing to secure their embedded code.**

Index Terms— **embedded software, application development, combat, threat modeling, penetration testing.**

1.0 Introduction

Software developers' greatest challenges in producing secure embedded code are rooted in the nature of the devices that run the software:

- They are resource-constrainedand have less "room" to compensate for CPU- or memory-robbing attacks. As a result, they are easily susceptible to denial of service attacks.

- Their performance can be slowed by cryptography. To speed performance, embedded developers do not include secure networking protocols on embedded devices as often as they do on their desktop counterparts.

- Their firmware can be changed. Knowledgeable users can swap out existing embedded firmware and replace it with an operating system of their choice.

- They are only intermittently connected to a network. Inconsistent network connections reduce the likelihood that security patches will be kept up-to-date, and increase the chance that the device will access an unsecure network.

- They are easy to steal due to their small physical size. In theory, an attacker could swap one embedded device for another and load malicious information into a system.

This paper will examine threat modeling and explain how it can be used in concert with secure development best practices, including automated source code analysis, peer code reviews, and penetration testing to both identify and mitigate embedded software threats.

2.0 Threat Modeling – A Brief Overview

Threat modeling is a security engineering activity that documents the key assets found in an application or system and purposely exposes risks to those assets in a thorough and disciplined manner. The goal of a threat model is to shine a light upon hidden security risks that may not be obvious or anticipated by the design team. This information can then be used to develop a risk management strategy and provide a roadmap for future security engineering activities.

By identifying an application's potential vulnerabilities, threat modeling helps development teams to understand and prioritize the array of risks for which the software is susceptible. With the results of a threat model in hand, development teams can ensure that they are concentrating their design, development and testing techniques on the risks that matter most.

2.1 Benefits of Threat Modeling

Threat modeling is one of the most powerful security engineering activities because it focuses on actual threats, not simply on vulnerabilities. A threat is an external event that can damage or compromise an asset

or objective, whereas vulnerability is a weakness within a system that makes an exploit possible. Vulnerabilities can be repaired, but threats can live on indefinitely or change over time. Threat modeling facilitates a risk-based software development approach by uncovering external risks and encouraging the use of secure coding practices.

In particular, threat modeling helps development teams to:

- Assess the probability, potential harm, and priority of attacks

- Prioritize security efforts according to true risk

- Shape an application design to meet security objectives

- Identify where additional security resources are required

- Weigh security decisions against other design goals

- Improve the security of an application by implementing effective countermeasures

- Understand attack vectors for penetration testing

- Understand the conditions under which an attack may be successful

By helping development teams to identify and understand potential threats, threat modeling provides the essential information needed to plan an embedded software security strategy.

2.2 Caveat to Threat Modeling

It is important to note that threat modeling is not an attack plan, a test plan, a formal proof of system security, or a design review. Threat modeling informs those plans and reviews by offering deep insight into the methods attackers could use to manipulate embedded software. Threat modeling is therefore a key contributor to design review and test planning, but should not be considered a substitute for those activities.

3.0 Creating a Threat Model

Developing a threat model is a team effort, but works best when the modeling exercise is led by a designer with security expertise. The following activity overview outlines an efficient and repeatable procedure for modeling threats to embedded software.

Step 1: Identify Security Objectives

First, the team must clarify the desired level of security. Is the goal to prevent any and all security breaches? Are certain attacks permissible? Preventing every possible attack may not be possible or cost-effective, so it is important to develop realistic objectives that balance security, cost and effort.

Step 2: Create a System Overview

Once its security objectives are clear, the development team should examine its software application and identify each asset of value. Assets of value are components that an attacker would value and which therefore need to be protected. Examples include:

- Data assets such as credit card numbers

- Technology assets such as intellectual property or content under Digital Rights Management "By helping development teams to identify and understand potential threats, threat modeling provides the essential information needed to plan an embedded software security strategy."

- Soft assets such as business reputation and customer trust. Certain attacks, such as defacement, can have a minor impact on hard assets but can dramatically reduce customer confidence in an organization's ability to develop a reliable, trustworthy product.

Step 3: Isolate and Decompose the Device's Software Design

While product developers are normally concerned with use cases, a threat model encourages the team to think about abuse cases. An abuse case is an attack scenario in which a malicious user wishes to abuse, rather than use, a system. The threat modeling process helps to generate abuse cases by "decomposing" a device's software design to isolate the area's most susceptible to abuse.
When brainstorming on abuse cases, consider:

- The data on the deviceand data in systems that can be accessed by the device.

- The input sources that could be used to attack the device software. These could include network data streams to the device operating

system, installed applications, GPS signals, and cellular voice/data entry.

- Physical challenges that could arise if the device finds its way into the hands of an attacker. For instance, how would you protect sensitive data if the device is stolen?

After enumerating the assets of value and decomposing the device's software design, a development team can generate a thorough list of threats that could negatively impact the device or system.

Step 4: Identify Threats

The goal of the threat modeling exercise is to identify as many threats as possible. To do this, development teams should use the "CIA method" and consider the events that would impact the Confidentiality, Integrity, or Availability of each asset.

Many devices, for example, reveal geographic information about the user. The "Google Latitude" function on a smart phone can reveal a user's physical location, and a log of "cardholder present" credit card transactions can identify a user's movements. Devices

with embedded software often log access to system resources. When compromised, this information can provide a blueprint of interesting and valuable information on the device.

Once a development team has identified any and all threats that could compromise the confidentiality, integrity and availability of its assets, it must consider the type of attacks that could be used to realize each threat. The most efficient way to identify potential attacks is to develop an "attack tree" for each threat.

An attack tree is a visual tool that documents threats and attacks for an asset, as shown in Figure 1. The threat is documented at the top of the tree and it is followed by a set of branches that represent potential attack methods. These branches are then further subdivided to identify the conditions or techniques that could be used in a successful attack.

In this example, the threat tree not only identifies the type of attacks that are possible when an attacker impersonates a user, it also lists the conditions and techniques under which a successful attack could take place. This information can be used in the next step of the threat model to identify the specific vulnerabilities within the embedded code.
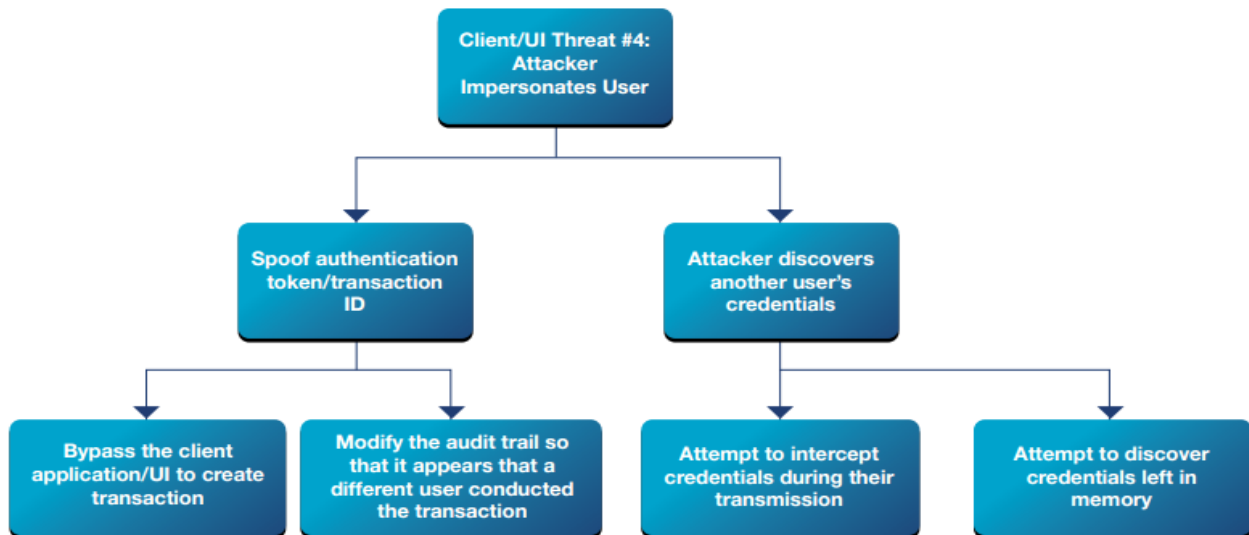

Figure 1: Sample Attack Tree for an impersonation threat

Step 5: Identify Vulnerabilities

A good threat tree will list all of the conditions under which an attack could be successful. Imagine that a threat model has highlighted that credit card information could be obtained from the system via a "man-in-the-middle attack" on a communication channel. In this case, the attack tree would show that the attack could be successful if credit card information

is transmitted over the data channel in clear text. If the development team finds that this condition is met in its system, it should develop a mitigation strategy to block the attack. If that condition is not met, an attack is not possible and the team can concentrate its efforts elsewhere.

At the end of this process, the threat model will comprise a list of vulnerabilities that can be used to plan an attack mitigation strategy.

Step 6: Repeat

Threat models are organic documents and should be revisited frequently. Conditions change, designs change, and the threat landscape changes. The DVD world, for example, provides an excellent example of the need for continuous threat modeling. When DVD players were first created, the keys for DVD Digital Rights Management (DRM) were included in the actual DVD player hardware. Hardware players were initially tamper-proof, but the introduction of software DVD players made it much easier for attackers to reverse-engineer the keys and breaks the encryption.

The original threat model for an early DVD player would have listed only the original threat: "DVD Content is Stolen", and its mitigation: "DVD content is encrypted, encryption keys are stored in tamper-proof hardware". With the introduction of software players, the threat model had to be updated to identify and mitigate the new risks.

Threat Modeling - Activity Summary Table

Input	Step	Output
• Business requirements • Security policies • Compliance requirements	**Step 1:** **Identify security objectives**	• Key security objectives
• Deployment diagrams • Use cases • Functional specifications	**Step 2:** **Create a system overview**	• Whiteboard-style diagram with end-to-end deployment scenario • Key scenarios • Roles • Technologies • Application security mechanisms
• Deployment diagrams • Use cases • Functional specifications	**Step 3:** **Isolate and decompose your device design**	• Trust boundaries • Entry points • Exit points • Data flows
• Common threats	**Step 4:** **Identify threats**	• Threat list
• Common vulnerabilities	**Step 5:** **Identify vulnerabilities**	• Vulnerability list

Figure 2 | Threat Modeling Activity Summary Table

4.0 Putting it into Practice: Identifying & Mitigating Vulnerabilities in Code

While threat modeling can uncover the broad threats and vulnerabilities of an embedded system, it cannot mitigate those threats. To do so, development teams must practice defensive coding, engage in frequent code reviews, and perform penetration testing.

4.1 Code Defensively
Defensive coding is a form of design that aims to ensure the continuing function of software and source code in spite of misuse or abuse. While a threat model can identify vulnerabilities due to design, a certain percentage of vulnerabilities will always result from coding flaws.

Developers often find that many of the vulnerabilities identified in the threat model result from only a handful of coding errors. One simple insecure coding technique that is performed repeatedly can contribute to dozens of vulnerabilities. Hackers frequently exploit the best-known vulnerabilities, so developers that code defensively and eliminate the most common coding flaws can substantially reduce the risk of a successful attack.

Moreover, threat modeling often uncovers threats that can only be mitigated through good coding practices. If, for example, an organization has identified a threat that requires a centralized input and data validation

strategy, it will require code-level fixes to accomplish the validation. These principles might include validating all input for length, range, format and type.

By following defensive coding practices – most notably, the use of automated tools to identify weak coding practices and uncover vulnerabilities – development teams can dramatically reduce the frequency and impact of bad code.

4.2 Automated Source Code Analysis

Automated source code analysis (SCA) tools provide a high return on investment for any software development organization by helping to eliminate bugs early in the development cycle. Industry estimates hold that the cost of addressing a code defect after a build is 10 times higher than addressing it during development. While automated programs do not remove the need for manual code testing, they can dramatically reduce the time spent on code reviews and focus manual tests on the most important and "hardest-hitting" issues.

Static analysis tools, for example, can identify hundreds – if not thousands – of coding problems. These include:

- Common vulnerabilities such as buffer overflows, uninitialized data, use of dangling pointers, injection flaws and known insecure APIs and libraries.

- Secure coding guidelines such as CWE, CERT, DISA and OWASP, as well as any custom checks or guidelines that would be unique to your code base.

- Reliability-related concerns such as memory leaks, memory allocation, resource management and more.

- Long-term maintainability concerns such as architectural violations, dead code, unused local variables, and other coding style best practices.

By incorporating automated static analysis tools, organizations can simplify existing peer review processes and automate a number of code review activities. Moreover, by running this analysis early in the software development process, developers can eliminate simple mistakes before they make it into the code stream.

In fact, static analysis tools are ideal for educating developers about the coding problems listed above. Most developers are not security experts, but source code analysis tools can help to inform and educate developers of the most common security issues. By examining static analysis results, developers can identify the frequent problems and, over time, make improvements in their processes to avoid them.

It is important to note, however, that static analysis can only identify specific coding problems. It is up to the development team to decide whether those problems need to be addressed. That decision depends on established trust boundaries and the costs/benefits associated with the repairs. Development teams can speed these decisions by consulting the threat trees established during the threat modeling process to determine whether the vulnerabilities represent true threats to the system.

4.3 Engage in Frequent Code Reviews

Security code reviews are critical in the development of secure code. They unveil vulnerabilities that are difficult to discover through testing processes since they examine the source code directly and review code paths deep inside an application. Through a focused and iterative approach to code review that consists of both manual and automated inspection, code reviews can be performed as often as every check-in to discover bugs before they make it into the build. These frequent code reviews not only identify additional vulnerabilities, they also allow developers to gain experience and learn collectively from their mistakes.

To perform an effective code review:

1. Identify code review objectives. Consult the threat model to prioritize risks and identify the most important vulnerabilities.

2. Perform a preliminary scan. Use both control flow and data analyses to step through logical conditions in the code, understand the conditions under which each block will be executed, and trace data from the points of input to the points of output.

3. Review for common issues. Scan embedded code for common vulnerabilities around data access, input and data validation, authentication, physical possession and replay attacks.

4. Review for unique issues. Consult the threat model and scan embedded code for vulnerabilities that may be unique to the particular system, device or application in question.

Code review should be started early in the software development process and repeated until the team is satisfied with the results or until a pre-established time limit has been reached. At the end of this process, the development team will have a set of prioritized vulnerabilities and inspection questions in hand that it can use to make future reviews even more effective.

4.4 Perform Security Testing

Security testing should be one of the final steps performed in an embedded software security project. Through a penetration test, development teams can gain confidence that their earlier design review, threat modeling and code review activities have hardened the software against attack. If teams have followed the security best practices outlined in this white paper throughout the development lifecycle, the problems that they will identify during this final stage will typically be minor and simple to remedy.

When an application is ready for a penetration test, leverage the threat model to improve the test plan. Use the threat model to determine attack vectors and conditions under which the attacks may be successful. Security vulnerabilities can be subtle, so be sure to consider all signs of a successful attack, such as an unexpected change to a file system, or unexpected network traffic.

Like a code review, a security test can also use both automated and manual tools. Automated SCA tools can be used to speed analyses, and manual testing techniques can be employed to both discover and address elusive vulnerabilities.

5.0 The Importance of Threat Modeling

Modern embedded systems are approaching the complexity of a traditional PC while introducing additional complexities related to connectivity and resource constraints. Through the use of key security engineering activities including threat modeling, code reviews, coding best practices, and security testing, development teams can detect and address security vulnerabilities in their embedded code quickly, efficiently and prior to product release.

6.0 About Klocwork and Security Innovation

Klocwork helps developers create more secure and reliable software. Our tools analyze source code on-the-fly, simplify peer code reviews and extend the life of complex software. Over 1000 customers, including the biggest brands in the mobile device, consumer electronics, medical technologies, telecom, automotive, military and aerospace sectors, have made Klocwork part of their software development process. Tens of thousands of software developers, architects and development managers rely on our tools everyday to improve their productivity while creating better software.

Security Innovation is an established leader in the software security and cryptography space. For over a decade the company has provided products, training and consulting services to help organizations build and deploy more secure software systems and protect their data communications.

7.0 Conclusion

Breaking Embedded Software News:

- "Google Confesses Android Security Breach, Rolls out Fix"

- "Sony Announces PS2 Bank Security Breach"

- "Microsoft Warns Xbox Live Users of Security Threat"

- "RSA Offers to Replace Tokens After Attack"

Embedded software becomes more ubiquitous and connected – powering everything from home appliances and cars to aircraft and mission-critical systems – organizations must take additional steps to ensure that the code produced is both secure and reliable.

ACKNOWLEDGEMENTS

REFERENCES

[1.] https://www.securityinnovation.com/
[2.] https://www.securityinnovation.com/solutions
[3.] https://www.securityinnovation.com/products/ secure-development-knowledgebase.html
[4.] https://www.securityinnovation.com/products/ encryption-libraries/arm.html
[5.] https://www.securityinnovation.com/company
[6.] http://blog.securityinnovation.com/
[7.] https://www.securityinnovation.com/security-lab/tools/
[8.] https://www.securityinnovation.com/security-lab/crypto/306.-ntru-technical-reports.html
[9] https://www.securityinnovation.com/services/software-assessment/software-security-threat-model.html