Threat Modeling

Or: How to Cheat in Exams

Philipp Kewisch

University of Applied Sciences, Wedel
Computer Sciences

Instructor: Prof. Dr. Gerd Beuster
Seminar Security Modeling 2014

Table of Contents

1. What is Threat Modeling?

Keeping software secure is not an easy task. It's nearly impossible to have all potential security breaches in mind while writing code. The focus is often on solving the problem, not on worrying about upcoming exploits in every written line.

Threat modeling is about creating abstract models of the system being developed, that aid finding potential security issues. It helps software architects design their software in a way that security bugs can be avoided up front.

Some security issues might be specific to the problem you want to solve. With an abstract model you can look at the problem in a way that will allow comparison to other secure systems and easily identify deficits.You will also be able to define security requirements to handle the common trade-off between usability and security. In the end it will help you deliver a better product.

The act of threat modeling happens naturally in situations where you might not expect it. Maybe you are late for class and are thinking about how to enter the room with the least arousal. Possibly you are bored and suddenly fantasizing about what you would do if someone threw a smoke grenade through the open window? You might have guessed it, even with these idle thoughts you are right in the middle of threat modeling.

This paper will give an overview on how threat modeling works: finding threats, addressing them and making sure they don't come back. Finally, this document will analyze a very specific threat model that will help you cheat in exams – or help you understand why its not worth it.

1.1. Strategies for Threat Modeling

There is not just one way to model threats, no grand scheme that can be applied once to squash all security bugs. Threat modeling is always about finding an approach to reducing security risks that gives the team confidence that what they have done using a reasonable amount of resources.[1]

Depending on the type and complexity of the system to be secured, different aspects can be emphasized in the threat model: assets, attackers and software. After a short summary of all methods, this paper will concentrate on the software-centric approach.

Focusing on assets entails creating a directory of resources used in the system, mostly those of value to the attacker. Each asset can then be checked for possible ways an attacker could threaten security. While this seems like the most intuitive approach, there is only a marginal benefit for the common software developer. Listing resources and discussing what components are security relevant assets merely results in a shallow list of starting points for security analysis. The methodologies for deriving security threats are much more difficult to conduct using the asset-centric approach.

Concentrating on attackers also seems like a feasible approach at first. Alas, concluding possible security issues based on how an attacker might view the system has similar downsides as the asset-centric approach. Each attacker has a unique skill set and motivation, making it hard to capture all possible attacks. The likeliness of overlooking a threat is high, especially if the team evaluating the system does not consist of security experts.

A software-centric approach on the other hand focuses on scanning the software models that are often created even without the specific intent to threat model for security risks. Each individual model can be

1 McGraw 2006, 161–162

separately checked for potential security issues. Evaluating the software security becomes easier since the problem is divided into smaller units, resulting in a more secure overall system. This extra pass often has the pleasant side effect of creating a better software model. The components are considered from a different angle, which leads to a more comprehensive understanding of the system.[2]

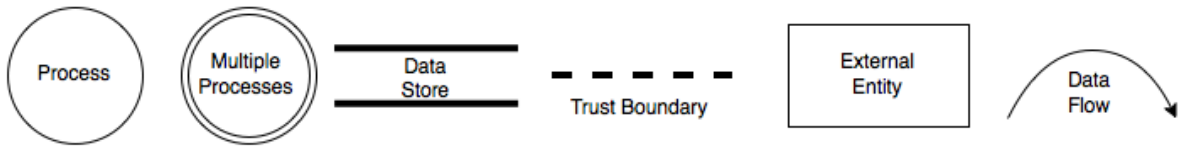
2. Modeling the System

Creating a new professional software system often presupposes creating diagrams, constructing a model of the system being designed. These diagrams can be used in threat modeling to provide a common understanding of how the components work, minimizing possible misunderstandings on the system's security.

This chapter will describe how diagrams can be used to support thread modeling and introduce the concept of trust boundaries for use within threat models.

2.1. Diagrams for Threat Modeling

An ideal type of diagram for threat modeling is a data flow diagram. Due to the hierarchical nature of the diagram, it is easy to capture how data flows through the system. It consists of one or more external entities that represent the input and output data of the system. In the midst, data stores and processes describe how the data is handled and contained within the subsection being described. Data flows are used to connect the other elements. Trust boundaries, which are described in the next section, use the boundary element of DFDs. The following symbols are most prominently used.



Other diagram types can also be used for threat modeling. For example, UML activity diagrams can equally well be used for threat modeling since they display similar aspects of the system.[3]

2.2. Trust Boundaries

An important concept in threat modeling is a trust boundary. Any time the level of trust and reliability changes, a trust boundary has been crossed. This can be an elevation of privilege on a singular system or receiving untrusted data from an external entity.

Drawing these boundaries can be done by analyzing the components in your system, for example different user ids, machines or network segments. Another approach is to examine the data being transferred, starting with either the most or least trusted source. Draw a boundary every time a subsystem of higher or lower trust has been reached.

Should a trust boundary be found on a single entity of the system, this is an indication that the entity should be split into more than one entity or described with a separate diagram. Similarly, if two entities can be considered the same from a security perspective, they can be consolidated in a simplified diagram.

The boundary should not be seen as a strong line. Untrusted data may cross a trust boundary for objects created as part of the execution process, for example caching. Nevertheless, data must be screened for

2 Shostack 2014, 34–43

3 Howard/LeBlanc 2002, 74

security before it can be further processed within the subsystem.[4]

2.3. Conclusion

Creating a model is not only required for good software design but is advantageous for threat modeling. Before moving on to find threats it should be ensured that the model matches what you have built or are building. It should be ensured that the model is complete, accurate and covers all security decisions.[5]

3. Finding Threats

There are numerous ways to find threats based on the models and diagrams previously created. Each concept has its advantages and disadvantages. This chapter introduces a specific approach called STRIDE, an acronym for the words: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privileges.

STRIDE describes families of attacks that help with identification. Time should not be spent arguing if a certain exploit is for example a spoof or tampering but rather with creating a mitigation strategy to avoid or eliminate the threat.[6]

3.1. Spoofing

Spoofing is an attempt to deceive by impersonating someone or something else. There are various categories of spoofs in the computer security context, a few of which will be described in this section.

One category is process spoofing where the user is tricked into sharing sensitive information with a malicious program that mimics a different tool. A backdoor program may also masquerade itself as a known process that is inconspicuous when an admin observes the process list.[7]

Similarly, file spoofing can be used to trick a legit program into reading data or calling library functions in a replaced file. Without extra checks, the program might consider the data safe to interpret or the library safe to call using elevated privileges or at least forcing the program into an unwanted action.[8]

Another category is machine-level spoofing. Different techniques like IP Redirection or DNS Spoofing can be used to avert data to a rogue host, intercepting potentially sensitive data or executing a man-in-the-middle attack. Phishing attacks also fall into this category. Here the user is tricked into entering personal data on a webpage that poses as a familiar site. The intercepted personal data can then be used for malicious activities.[9]

The final category to be mentioned here is spoofing a person or role. This is related to identity theft, where an attacker impersonates the target and uses that identity to access sensitive information or trick others into providing this information.[10] Some sites offer the concept of "verified accounts" to mitigate such spoofs.

3.2. Tampering

Tampering has overlap with spoofing to some degree. It entails modifying a resource in a harmful way.

4 Cornutt 2013
5 Shostack 2014, 24
6 Howard/LeBlanc 2002, 83–86
7 Dulaney 2011, 60
8 Graham/Olson/Howard 2011, 253
9 Jakobsson/Myers, 1, 74, 124
10 Cole 2002, 136

The resource is typically something on disk, on a network or in memory.

The most notable form of tampering applies to computer memory. If an attacker is able to modify the contents of memory used by a process with elevated privileges, the process might naively interpret the data, causing different code paths to run which are more favorable to the attacker.[11]

Tampering can also apply to files, an attacker can modify anything they have or gain write access to. This applies both to local files and remotely included files on the network. Should an attacker gain access to a content delivery network for javascript files, they can tamper with the files and add mechanisms to redirect sensitive information of sites using the CDN to a different machine.[12]

Network tampering can also occur on a lower level. An attacker may be knowledgable enough to modify or prevent wireless network data in the air, causing misinterpretation on the other end. This is an area that software developers often neglect. If a smaller company has no network operation specialists, this kind of attack should definitely be taken into account.[13]

3.3. Repudiation

Repudiation is the act of denying responsibility. In the security context this often equates to covering up harmful operations or the opposite, injecting false attacks into log files to make analysis more challenging.

The target of repudiation does not have to be the attacker. Instead, the attacker might have used techniques to make it look like an employee or user submitted to harmful activity. There are also situations that do not involve an attacker at all. Maybe user didn't perceive that he clicked on the button, because the product's interface lacks user friendliness.

Solid and safe logging helps mitigate these kinds of security threats. In the long run, this might also improve customer experience, as its easier to retrace the user's actions and understand if the complaint is honest or deceptive.[14]

3.4. Information Disclosure

Providing information to someone who is not authorized to see it is the core aspect of information disclosure. In the IT area this usually means finding technical details about a system that are not meant to be exposed, either by incorrect permissions, overly verbose error messages in logs or drawing conclusions from (network) data analysis.

The following example shows how finding out technical details about a system and verbose log messages can lead to information disclosure. The previous section about repudiation suggests logging to mitigate security threats. Alas, adding too much information to the error message may lead to an information disclosure risk.

Imagine a system logs invalid password attempts together with a similarity index between the actual password and the password entered. If random passwords are entered and/or similarity to the last entered password is to large, the system would disable the account. Simple typos would not affect the account status. What may seem like a feature to make access easier for the user could lead to an information disclosure risk.

The feature has flaws on multiple levels. First of all, an attacker with no additional access could randomly

11 Blunden 2013, 339–340
12 Michalakis/Soulé/Grimm 2007
13 Hong Kong 2010, 9–12
14 Shostack 2014, 68–70

try one password for a group of accounts multiple times. A lot of accounts will be banned because there is no similarity, but some accounts will continue to work. The attempted password can then gradually be changed for those accounts until they are disabled. Either the attacker will find a password immediately, or can continue the attack when the accounts are available again.

Also, if the attacker gains access to the logs through a different security loophole, he can use the similarity index to increase the cracking speed. While this may be a constructed scenario, information disclosure is often an underestimated risk and should be taken seriously.[15]

3.5. Denial of Service

A denial-of-service attack encompasses taking up all resources required to provide a service. This may be done on the machine-level by depleting the system memory or slowing down processes by keeping the CPU busy.

Also, sending incorrect input to a process might cause it to spend more time validating the input, logging errors about it and notifying the user than it does providing its actual service. This in turn could fill up data stores with logging information which in turn may cover up other attack forms or simply render the service unusable.[16]

3.6. Elevation of Privileges

Elevation of Privileges entails gaining privileges an attacker is not authorized to have. This could mean that the attacker gains administration rights with the means to disrupt the service or circumvent security measures. Common ways to elevate privileges are through process corruption or circumventing authorization checks.

A process can be corrupted by a number of data manipulation techniques. The attacker then takes control of the data flow and can either disable security measures fully or receive unauthorized privileges. If the program or site doesn't check for authorization on every code path that executes privileged code the attacker can exploit the circumstance and elevate privileges.[17]

These attacks are not limited to the exposed attack surface but can also be reached indirectly. An attacker may gain access to the system using a different exploit and then continue with an elevation of privilege attack to fully control the target. Therefore it is important to check all code paths that can lead to privileged code being run.[18]

3.7. Applying STRIDE to the Model

With security it may feel like there is always one more threat to find, so its important to be able find exit criteria that allow you to move on to the next stage. For simple systems it may be enough to determine one or two threat for each STRIDE category. The next possibility is finding one threat per category for each element of model diagram. This means substantially more threats to find, although it does not necessarily cover all threats.

The downside to finding one threat per category is that you either end up with a lot of threats that are not relevant at the end or have trouble finding threats for certain elements. The solution is STRIDE-per-element, a table that defines which categories should be applied to which elements. After all, its not very

15 Howard/LeBlanc 2002, 701
16 Mirkovic et al 2005, section 5.6.1
17 Dulaney 2011, 3
18 Graham/Olson/Howard 2011, 229–230

common to have an elevation of privilege within a data flow. Instead, the process it leads to might be susceptible to an elevation of privilege attack. The following table shows how STRIDE-per-element should be applied.

Element	S	T	R	I	D	E
External Entity	✓		✓			
Process	✓	✓	✓	✓	✓	✓
Data Flow		✓		✓	✓	
Data Store		✓	?	✓	✓	

Regarding the question mark, data stores are normally not sensitive to repudiation threats. Then again, logging is often used as a mitigation for repudiation threats on other elements and these logs are held in a data store. Access to logs could cause information disclosure. Threats that are found on the way that don't fit the STRIDE-per-element table should not be ignored and processed as any other threat, they are just not as common.

No matter which approach is chosen, there is no guarantee that all threats have been found during this process. Securing products is not a one time effort but instead a cycle that needs repeating, for example after a certain software milestone has been completed. The next chapter will cover the next phase of this cycle, addressing the found threats.

4. Addressing Threats

In order to continually deliver a secure product, threat modeling does not end when threats have been found. Tracking threats helps avoid security issues in further iterations of the product or when the surrounding conditions change.

This section shows how threats can be tracked and addressed. There are situations where not all threats can be avoided, therefore tactics for risk mitigation are also important and will be laid out in this section.

4.1. Processing and Tracking Threats

The primary form of tracking threats is the use of tables and diagrams. Diagrams have been discussed in details above and provide a basis for threat modeling in combination with software modeling. In addition, tables and lists can be used to track identified threats for future use. Three major ways the tables can be organized are tracking by diagram element, by threat type or by order of discovery.[19]

Each table should also reference a bug filed in the product's bug tracker. Some bug tracking solutions (for example bugzilla) also allow for fields and queries so that threat modeling data can be managed directly from within the bug tracker.

Another important aspect to track is assumptions. When threat modeling, you often make assumptions on what threats might be acceptable, how the system works and handles security situations. If you are tracking these assumptions, you can follow up and make sure these don't become exploitable, especially when software changes.[20]

When all threats have been listed and bugs have been filed, the modeling phase of threat modeling is completed for the time being. Actually fixing the issues or applying trade-off strategies is nevertheless

19 Shostack 2014, 133

20 Shostack 2014, 135

important, but can be handled within the standard software development process. A further iteration of threat modeling can be applied at a later stage to verify that the issues were fixed and no new security threats have come up.

4.2. Tactics for Mitigation

When addressing threats, its important to begin with mitigating the first order attacks. These attacks are those you have previously found during the modeling phase and are commonly blocked with new features. An attacker will attempt to circumvent each new feature, so this also applies to such mitigations. These attacks are called second or third order attacks, or generally, higher order attacks. Think of it a little like playing chess: each move will cause the attacker to search for a countermove that you should be prepared for even before making your move.[21]

The following subsections will inform about mitigation tactics for each STRIDE threat type. The goal is not to provide a complete reference on all possible mitigations as the actual solution is mostly dependent on the system being created and the technologies being used. Therefore the following subsections will merely provide examples on how a certain scenarios can be mitigated.

4.2.1. Mitigations for Spoofing

As spoofs are generally only possible for programs running at the same or lower level of trust, the ground rule for mitigating spoofing is to trust only code running at a higher level of trust. All other inputs or control flows should be validated using some form of authentication.

Note that authentication should not be mixed up with authorization. Both terms are commonly used for a possible consequence of each, providing a prompt for a username and password. Going back to the definition, authorization defines a set of rules that determine which actions the user (or component) is allowed to perform. Authentication on the other hand validates a user's or component's identity. [22]

Spoofing is often an issue over an unsecured network. The obvious solution is to use cryptographic trust mechanisms to validate that the machine making the request is truly the correct machine. Authenticating a person is more complicated, as simple mechanisms like usernames and passwords can easily be compromised. The use of cryptographic keys improves security and with the right certificate authorities and validation mechanism its possible to make sure that the user identity is valid. Of course the private key might be compromised as well, which would qualify as a second order attack.

Machine-level spoofing can be done through loading rogue libraries in place of authoritative ones, which can be mitigated by using full paths or validating digital signatures.

4.2.2. Mitigations for Tampering

Mitigating tampering is usually a matter of validating data integrity. Three possible ways to address tampering are relying on the operating system, using cryptographic mechanisms and auditing with logs.

The operating system can provide a sufficient amount of protection against tampering, as long as everything is set up right. File permissions need to be set correctly to keep the attacker from overwriting configuration and security information trusted by the program.

The aforementioned cryptographic mechanisms mainly consist of hashes and digital signatures. The hash allows to verify that a digital object has not been tampered with, while digital signatures allow to do so

21 Shostack 2014, 130–131

22 Ciampa 2005, 267

on a much larger level. Tampering on a network level can be prevented through the use of TLS or IPSec, which makes use of digital signatures.[23]

Logging on the other hand is a mechanism that does not prevent tampering per se, but allows you to recover from attacks by rolling back the logged changes.

4.2.3. Mitigations for Repudiation

Handling Repudiation is not only important from a technical perspective. Business transactions rely on non-repudiation to avoid loss of profit and fraudulent transactions. On the other hand, repudiation can also be a requirement, for example when sending user-encrypted, possibly unlawful content. Plausible deniability is required to avoid taking responsibility and fighting legal battles.

The traditional approach for mitigating repudiation is the use of logs. It is important to carefully select the information being logged to avoid information disclosure and irrelevant information that might make log analysis more complicated. Nevertheless, all information required to trace the actions being performed needs to be contained in the logs. Analyzing the collected data can also help prevent fraudulent transactions, for example using machine learning of known fraudulent attacks to detect future attempts.

Sometimes logging is not enough, the other party may claim that the logs themselves were forged. In this case, digital signatures using public key cryptography can be used as proof that certain actions have been performed.[24] This in turn requires trust in certificate authorities that have proven the identity of the user. Still, the intent of the signature must be made clear to both parties. There may also be some legal implications around the acceptance of digital signatures. To ensure that a user action is non-refutable, always consult legal council.

4.2.4. Mitigations for Information Disclosure

Information disclosure can occur over a network or on a storage device. It is not restricted to the content of the message being disclosed, merely the existence of a transferred message can also disclose information an attacker should not have access to.

In the simple case where the system controls access to all data, a permission system can be used to prevent unwanted access. Data that is sent over a network must either be encrypted directly or the communication channel the data is sent over must be secured. Encryption can also be used analogous for stored data, either encrypting the data itself or the container it is stored in. If the existence of a transferred message needs to be concealed, onion routing can be used. With this method of transfer, the message is encrypted multiple times and transferred over various channels, creating hard-to-trace communications.[25]

Each encryption requires planning as to who needs to have access to the data, which requires thought on key management and distribution. Also it should be noted that simply encrypting the content does not ensure data integrity. If an attacker replaces the encrypted password of an administrator with his own encrypted password, the password itself is not disclosed. Nevertheless, using the replaced password opens the system to the attacker and can reveal other confidential information.[26]

23 Howard/LeBlanc 2002, 119, 290–296
24 Graham/Olson/Howard 2011, 3
25 Peng 2014, 115–117
26 Shostack 2014, 153–154

4.2.5. Mitigations for Denial of Service Attacks

The commonly known form of a denial of service attack is the distributed brute force attack, also known as a DDoS (Distributed Denial of Service) attack. A large number of requests are made to the target system, exhausting CPU, memory or bandwidth.[27] A possible strategy against such attacks is to ensure that the possible attacker can receive data that is sent back. This will stop an attacker making an abundant number of fire-and-forget requests from different nodes.

Another clever form is a denial of service attack, where a small amount of work on the attacker side is amplified, causing the target system to perform a task that takes more resources to execute. An approach to mitigate this is called *proof of work*. With this concept, the client must prove to the server that it has taken more resources to prepare the request than it will take the server to process it.[28] This works fairly well for small attacks, but an attack from a large bot-net will not stop at this.

Generally it is important to understand which resources an attacker might consume and instate a mechanism to limit the availability per client. While traditionally these resources are CPU, bandwidth, disk or main memory, depletion of other resources should also be considered. An attacker might keep the IT department personnel busy by raising alerts in the monitoring system. In the meanwhile, the attacker can proceed to the real attack, having enough time to cover up his actions before the logs are analyzed.[29]

4.2.6. Mitigations for Elevation of Privileges

Mitigating elevation of privilege attacks presupposes designing a safe and solid authorization system. General prevention mechanisms like limiting the attack surface or designing the system into smaller components that are easier to secure will help avoid such threats. Each component should make sure that all input is validated, distrusting by default. Instead of attempting to find a complete list of bad combinations, create a whitelist of values to allow.

A further strategy that should be followed is called *layered defense*. Instead of letting an attacker gain the highest privilege level through a single bug, provide multiple layers of security that each needs a unique exploit to overcome.[30]

4.3. Using Patterns for Mitigation

Software designers use software patterns to solve recurring problems in a maintainable way. The same concept can be applied to security. A security pattern provides formalization of security concepts using a context, problem statement, structure of the system it is applied to, system dynamics and a possible implementation. The range of available patterns is high, therefore this section will only provide a simple example.

The *access matrix* pattern describes who is authorized to access system resources and which operations can be performed. The context is a system where certain resources have higher value for the users. The system requires a concise mechanism on which user is allowed to access specific resources that doesn't suffer from confidentiality or integrity problems. The system must provide independence, flexibility, modifiability and security. The structure consists of a subject, an object of protection and a right that the subject is authorized for. System dynamics describe the operations that can be applied to the system, for example adding a rule or creating a user. Common implementation approaches are using access control

27 Mirkovic et al 2005, section 1.1
28 Mirkovic et al 2005, section 5.6.1
29 Shostack 2014, 155
30 Nahari/Krutz 2011, 229–230

lists or assigning capabilities.[31]

A different but related form of using patterns is the attack patterns database CAPEC[32]. Attack patterns are not software patterns, but rather describe possible attacks. These patterns are used especially in threat modeling using the attacker-centric approach, but since most patterns also include defense strategies that can be used in a software-centric approach.

4.4. Tradeoff Strategies

After the risks have been identified it is important to decide for each risk if it should be avoided, addressed, accepted or transferred. To do so, you must decide the level of the risk, what needs to be done to address the risk and how are these actions going to be executed. Depending on the number of risks this might be a time constraining activity. In such situations it may be advisable to skip these questions and just address the risks.

Avoiding risks essentially means avoiding the code that exposes the risk. This means either changing the design so that the risk vanishes or not building the feature because you have found that the effort to address the risk is higher than the win of adding the feature.

Addressing risks can similarly be done through design changes, such as adding mitigation code and components. It can also be done through operational changes like adding a firewall. Note however that changes made through this outcome should themselves be subject to identifying risks, as each new component can be subject to attack.

A further outcome can be accepting the risk. This decision is more than just identifying the problem and making no changes (that would be ignoring the risk). All consequences must be determined and be deemed acceptable, which entails both monetary setbacks such as legal costs as well as nonfinancial consequences including losing user trust. Risks concerning users cannot be accepted, but must instead be transferred through adequate means discussed further on. Accepting risks is often done when the probability is very low or the consequences are minor.

Fully addressing all risks is illusory, therefore some risks must be transferred to the customer or user. This is often done through legal mechanisms such as license agreements, but can also be done by informing the user within the user interface.[33]

4.4.1. Addressing or Avoiding Risks

Once the decision has been made to address or avoid the risk, different approaches can be applied. This includes eliminating the feature, changing the design or applying standard or custom mitigations.

4.4.1.1. Eliminating the Feature

Eliminating the feature is a simple approach to address or avoid a risk. This may not mean fully removing all traces of it, reducing the feature's reach is also a form of elimination. For example, when viewing a remote administration system accessible through both the internet and the company intranet, eliminating the feature of internet access can address possible threats. Nevertheless, it may not be possible to do so due to client requirements or hampering core product functionality.

31 Fernandez-Buglioni 2013, section 6.2

32 Common Attack Pattern Enumeration and Classification, https://capec.mitre.org

33 Westfall 2009, 293–295 and Paul 2014, 26–27

4.4.1.2. Changing the Design

Another tactic is changing the design. In this case, when avoiding the risk, the software must be designed so that the feature is still available, but the component being threatened is no longer needed. When addressing the risk, new components or other features may be added to deal with the risk.

Such design changes can be done iterative and comparative. The iterative approach entails gradually changing the system, either removing components or shifting the trust boundary to ensure that the risk cannot be taken advantage of. The comparative approach often requires more work, but is especially useful in early stages of development. One or more competing designs are created and compared to the original design to determine if the risk has been addressed. However, a new design requires further analysis to ensure that no new risks emerge.

4.4.1.3. Standard Mitigation Technologies

If the design cannot be further changed, standard mitigation technologies can be used, which means using known and proven effective mechanisms that have had sufficient testing. Such mitigations can be platform-provided, developer implemented or of operational nature. Instead of choosing one, a combination can and often should be used.

Platform-provided defenses are those provided by the operating system or development framework being used. These facilities have the advantage that the amount of security testing required is lower than usual, because both attackers and other contributors or vendor security teams constantly examine the system providing for a more secure platform.

In most cases, platform-provided defenses do not provide protection against all threats. The developer must implement additional security measures to mitigate risks by adding new features such as a memory protection system to address tampering attacks.

If developer-implemented or platform-provided mitigations do not suffice, operational mitigations can be deployed. With the help of procedural guidelines for deployment, high level protection such as packet filtering and anti-virus monitoring, security can be improved without coding changes.[34]

4.4.1.4. Custom Mitigations

The final approach is developing a custom mitigation, highly specialized for the system being protected. Following this approach is often time consuming and the result is hard to verify. Months worth of work may be demolished over just a few hours when the result is presented to a group of security experts. Careful definition and planning is required to ensure the method is not susceptible to its own attacks and is free of design flaws. Sharing the design with the public or offering a bounty to break the system provides for useful feed back even in early stages.[35]

4.4.2. Prioritization Approaches

As the amount of risks increases exponentially with code complexity and size, it is possible that not all risks can be addressed at once. There are multiple ways to prioritize the mitigations, each with their unique advantages and disadvantages.

The first category of approaches is fairly quick to implement, but only applies to very simple systems. For non-critical systems, risks can be chosen to be ignored given sufficient monitoring allows making conclusions to how the breach occurred and how it can be fixed. Another approach in this category is to

34 Shostack 2014, 174–176

35 Shostack 2014, 176–177

concentrate on security risks that are easy to fix which brings satisfaction among developers, but doesn't hold as a long-term strategy.

A more advanced technique is the use of a "bug bar", which allows ranking threats with more precision. Each issue is classified based on a previously defined set of criteria, created in advance by the team working on the system. For each severity level (e.g. "Critical" to "Low") and each STRIDE category, criteria are set that describe situations where the level applies. For example, "All write access violations in remotely callable code" could be considered critical, while "All write access violations from remotely authenticated users" are categorized as important. The "bar" comes into play when planning development. When a product release is imminent, the bug bar will be high and only critical bugs should be fixed. The more time there is, the more likely it is to fix low priority issues.[36]

Aside from technical reasons to fix security issues, there may also be an important business aspect to consider. One technique that takes business aspects into account is called Delphi Ranking and is based on the Delphi method: Using a common set of ranking criteria, individuals communicate their opinion on each risk only to an assistant, to avoid being influenced by others. The experts can come from different departments, including those that are not skilled with software development, such as business operations. The assistant then determines the risk based on the results. This method is usually used to determine risk priority from a business vantage and should be used in combination with other methods.[37]

Risks can be also be quantified by probability and impact for the company, issues with high ranks need to be fixed first. While the monetary impact may be easy to determine, impact to trust and calculating probability are difficult to establish. The focus is again put on the attacker, which has been determined suboptimal earlier in this paper: even well-designed systems can be broken with simple unconventional means, if they have not been though of during the design process.[38]

4.4.3. Risk Acceptance and Transferral

When accepting a risk, the decision can be made based on business policy or the user is given the option to explicitly accept the risk.

Business risk acceptance uses both business policy and laws to make the decision if a risk can be accepted. The company might chose to define business policy that all information is open and accessible, but is confined by privacy laws and other regulations so that certain information must be secured. Another example of business policy would be that the company is selling a product intended to be used in an environment that must not be connected to the internet. With appropriate disclosure, the customer is responsible for securing the environment and the risk has been transferred.

User Acceptance allows the company to transfer the risk acceptance to the customer or user. Being too strict in security always leads to a diminished user experience and there may be legit reasons for the user to accept a risk the company may have deemed hazardous. A prominent example is visiting sites with self-signed certificates in a web-browser. The browser has determined that the site being accessed is not trusted, but still wants to allow the user to access the site if he knows what he is doing. The situation is explained and it is left to the user to decide if he wants to access the site nevertheless.[39]

5. Verifying Threat Elimination

When all threats have been found and mitigated, work is not done. Just as in the software development

36 Microsoft 2012
37 Paul 2014, 214
38 Paul 2014, 216
39 Shostack 2014, 184–185

cycle, it must be verified that the threats have been eliminated. Aside from automated testing, manual quality assurance is also advisable. Another important step that is often forgotten is checking third party software to ensure that it addresses the threats that have been determined.

5.1. Automated Testing

Integration into automated testing ensures that no security regressions occur. This does not simply mean to write a test for each feature created. Additional tests that specifically test the security aspects are recommended.

The first kind of test case to write covers the immediate danger of a particular threat. It should attempt to exploit the component just as a real attacker might do so. This is much easier to do when patching a known exploit because it can be ensured that without the patch the risk is present, while with the patch the security hole is closed. When working on a new feature it might be a good idea to make local changes that would expose the risk, write the test case to check, then remove the local changes to ensure the test case is correct. Otherwise there will be a false sense of security that could lead to trouble later on.

On top of the just created tests, you should venture into a second kind of test that target the mitigation itself. Adding a test that attempts to bypass the mitigation adds another layer of protection against regressions. Further layers of mitigations demand further layers of test cases.

During regular testing, there are a variety of ways a feature might break. Similarly in security, an attacker will find different methods to exploit a component. This should be kept in mind while writing the test case. If in doubt, multiple tests should be written that ensure a feature cannot be exploited, each with their own additional mitigation tests.[40]

Another commonly used testing approach is penetration testing, where attempts are made to penetrate the system from the outside, possibly by using knowledge of how the system was designed or the code was written. However, penetration testing must not be seen as a replacement to threat modeling. Penetration testing will help to find additional security bugs, but it will not uncover design flaws until after the fact, where it may be expensive to make major changes. Threat modeling provides a more organized approach at a fine grained level, avoiding security bugs before they can be exploited.[41]

5.2. Checking Third Party Code

In many cases software is not written without external components, either in binary form or from source. These components are often forgotten when checking for risks, because it is assumed that the external component is secure. When introducing third party code, you should ensure that it has a sufficient level of security and your product's security requirements are fulfilled.

First of all, a software architecture diagram for the external component should be created. Even if the component already provides one, it should be ensured that it contains everything you need for threat modeling, foremost trust boundaries. When creating this model, start with those parts that are exposed, e.g. network listeners or file I/O and make sure the trust boundaries are set. If the component has any dependencies, these should be examined too. If in doubt, treat any data going in or coming out of the external component as untrusted.

With the software model created, you can apply standard threat modeling techniques to check if the third party code behaves as you would like it to. If security issues have been found, the producer of the third party code can be instructed about the issue. If the source code is available you can fix the issue

40 Shostack 2014, 187–191

41 Engebretson 2013, 4

yourself or apply operational mitigations to protect yourself against the issue.[42]

5.3. Quality Assurance in Threat Modeling

At some stage between feature completeness and a release the threat model should be verified to determine if thread modeling was successful. It should be ensured that the model matches reality as close as possible, all threats are addressed, avoided, accepted or transferred. If an issue tracker is used, all issues about threat modeling should be closed.

During this verification stage the architecture diagrams should be studied, looking for any inconsistencies. If there have been major changes on the way and the diagram was not updated, this is a good time to reevaluate the threat model to guarantee nothing has been missed.

Another pitfall to look out for as a QA engineer is developers wrongly believing that data has been validated. While for a certain purpose this might be true, attackers often benefit from being able to enter data that is correct for the purpose but could in turn uncover bugs that compromise security. An email validation form might accept special characters in the name part, wrong processing could lead to SQL injection attacks. A common remedy is to transform the input before it is further used. For example, when displaying user-contributed content on a website, enforce a safe subset at the presentation level, regardless of when and where the input was saved.

To make work easier for testers, it is also useful to document any assumptions made during the development process. QA can then test these assumptions by checking what would happen if these assumptions are not true.

If not all security issues have been closed, they should be triaged just like other software issues, using the severity as a key to determine if it can be postponed or not.[43]

6. Threat Modeling by Example: How to Cheat in Exams

While threat modeling is mostly applied to computer systems and software, it can also be applied to any area in life. As this is a seminar paper for my university I have chosen a very special example to demonstrate how important threat modeling can be: cheating in exams.[44]

The roles in this example can be looked at in various ways and since its not a technical system being modeled some creativity may be required, but the basic process of threat modeling can be applied to this system just as well.

The tricky part about this example is figuring out who the attacker actually is and what system should be modeled. Are the instructors attacking the cheating of the students, or is the cheating student attacking the system of the exam itself.

From a student perspective, the system to be modeled is the student himself as well as the work area around him. Possible attackers are instructors or assistants, who are attacking the attempt to cheat by trying to uncover it.

On the other hand, the instructors aren't doing anything wrong, so why should they be attackers? Turning the model around, the system being modeled could be the exam itself. Instructors and Assistants are components attempting to guard against attacks and the students are the attackers attempting to

42 Shostack 2014, 192–194

43 Shostack 2014, 195–202

44 It goes without saying that I do not condone any form of cheating in exams. I have not and will not use the threat model created here to circumvent any restrictions instated by my university. I will leave that exercise to the reader.

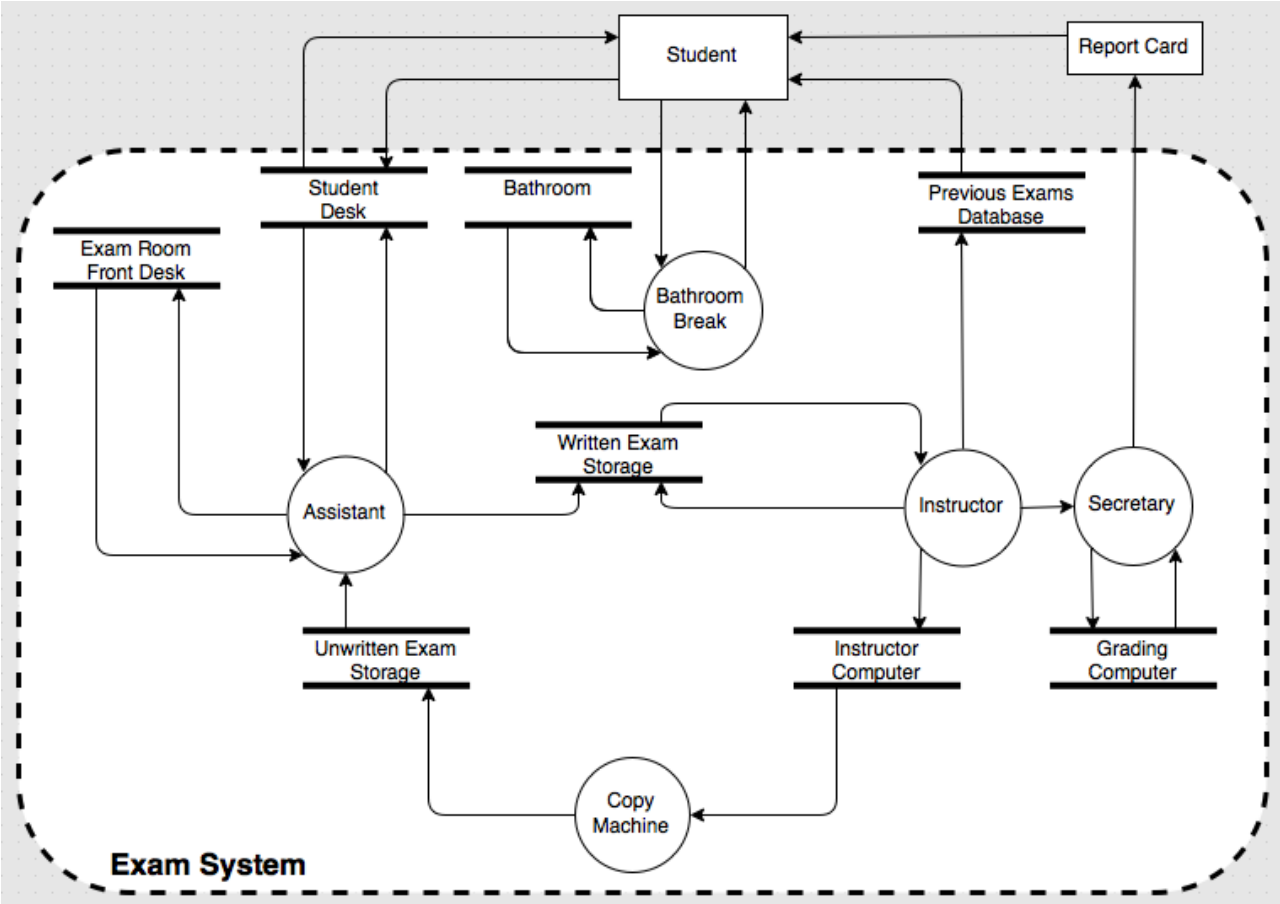
circumvent security measures.

For the sake of simplicity I will be modeling the system from an instructors point of view. Students reading this paper should consider this a glass box to be used for penetration testing. Knowing more about the threat model instructors use to protect against attacks helps find weaknesses and exploit them for your means.

6.1. Modeling the System

As mentioned in section 1.1, we will be concentrating on the software-centric approach. In our example, the software is much more a whole system of components that contribute to setting an exam. To model the system, we need to figure out which elements the system contains, how data flows between the elements and which trust boundaries exist.

In our example, the data can be thought of as the exam questions, which flow through various processes: the assistant retrieving and handing out the exams, the copy machine printing unwritten exams from the instructors computer, the instructor creating and grading exams, the secretary entering exam results into the grading computer and a potential bathroom break a student could be taking. There are also various data stores, which map to places the exam questions and results are placed during the procedure of the exam. Students and the final report card are considered external entities because they interact with the system as a whole. The following illustration shows our system components as a data flow diagram. For simplicity, the processes are reduced to the actor that control them.



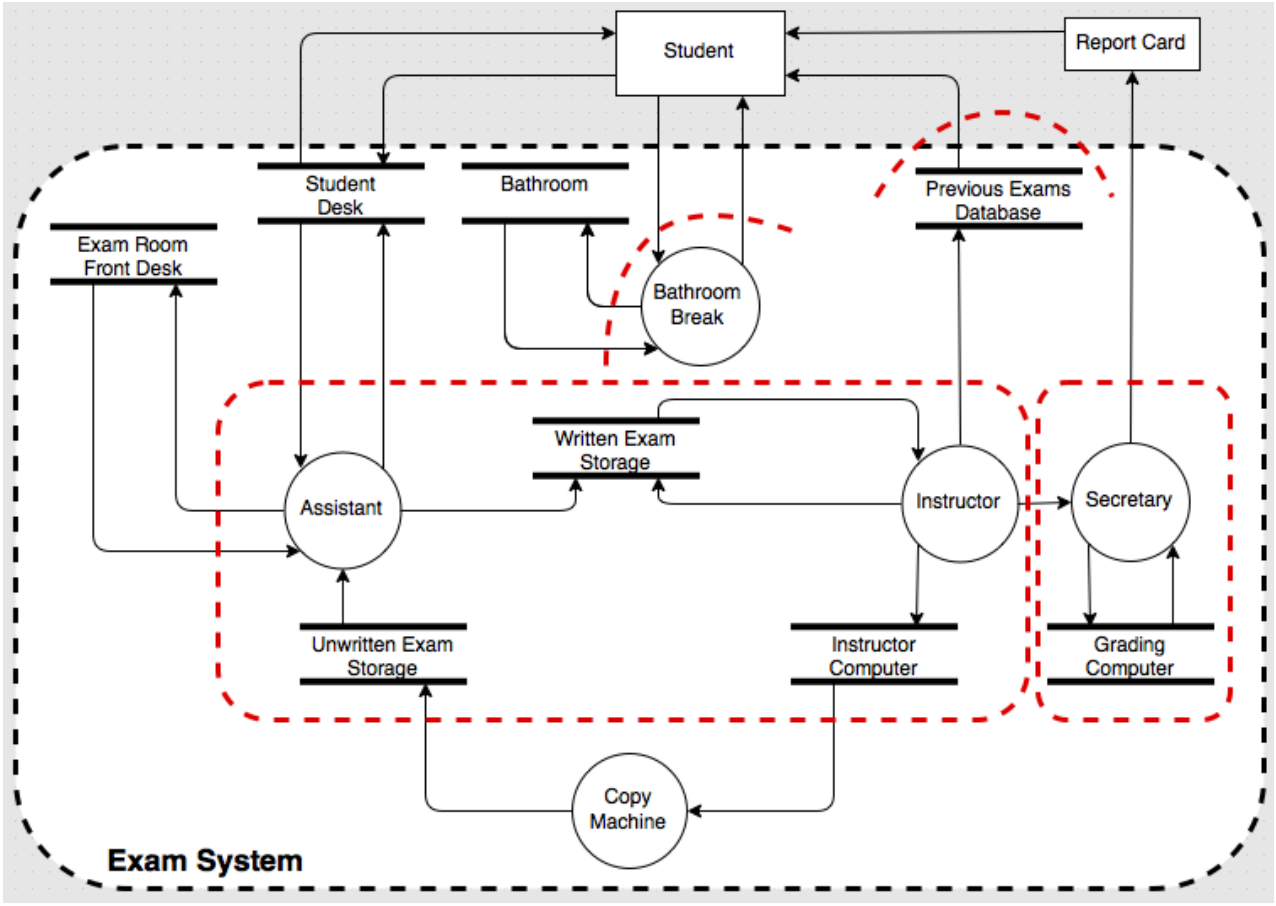
Next, we learned about trust boundaries in section 2.2. For this example it seems easier to determine the trust boundaries by following the data flow. Starting from the student perspective, the student has

unrestricted access to the student desk and read-only access to the previous exams database.

During the exam, the student may also take bathroom breaks, where hints and solutions may be stored and possibly shared between students. While the bathroom break itself may be a controlled process, the bathroom is publicly accessible and requires a trust boundary.

A more obvious trust boundary is the one between the student desk and the assistant. Given the previous exams database must only allow read-only access, there is also a trust boundary between the student and the exams database, even though full access to the exam database doesn't have an immediate advantage to the student.

The copying machine may be subject to attacks and is therefore not within the same level of trust as the instructor computer. Also, the grading computer and the secretary using it need to be separated to allow for non-repudiation. The following diagram shows the previously described boundaries.



6.2. Finding and Addressing Threats

Now comes the brainstorming part, determining risks for each threat category. As mentioned in section 3.7, we need to find at least one threat per category or better yet use STRIDE-per-element. Our diagram holds 5 processes, 8 data stores, 23 data flows and 2 external entities which means a minimum of 127 risks for STRIDE-per-element.

To avoid flooding the next few pages with risks, the following tables will contain just a few examples per category. You are likely to find many more cases for your own university. For additional brevity we are going to list the risk together with its mitigation strategy and the approach used for mitigation.

6.2.1. Spoofing

As a reminder from section 3.1, spoofing is an attempt to deceive by impersonating someone or something else. Going by STRIDE-per-element, spoofing applies to processes and external entities. Here are a few example elements that can be attacked:

Element	Risk	Action	Mitigation Strategy
Student	Student can be replaced by twin brother or ghostwriter, impersonating the student.	Address	Check ID card closely before each exam.
Copy machine	Remote access to the copy machine allows for various spoofing attacks, e.g. DNS redirection.	Avoid	Disconnect copy machine from network.
Assistant	Student impersonates an assistant and gains access to the unwritten exam storage, retrieving the questions before the writing the exam.	Accept	Low risk. Detect abnormal grade averages and changes in student processing time.

6.2.2. Tampering

Tampering entails modifying a resource in a harmful way and is explained further in section 3.2. With STRIDE-per-element, tampering can be applied to processes, data stores and data flows. Tampering with exam questions is not in the focus of this category since its easily detected. Instead you will find a few other example elements that can be attacked:

Element	Risk	Action	Mitigation Strategy
Grading computer	If grades database is modified through tampering, report cards will be printed with false results.	Address	Put an air gap between the grading computer network and the remaining university network.
Student desk	Student has access to the exam desk even before the exam and can stash cheat slips.	Accept	Impossible to prevent all such attacks. Use policy to inform students about consequences. Monitor for violations and document attack methods.
Instructor → Secretary	When exam results are transferred via email in clear text, a student may intercept the email and send modified exam results.	Address	Mandate the use of signed and encrypted email for communication between employees.

6.2.3. Repudiation

As you have read in section 3.3, repudiation is something that certain students do on a daily basis: the act of denying responsibility. STRIDE-per-element plans for searching this kind of threat in processes, external entities and under certain circumstances also for data stores. The following table lists two situations, although most situations can be summarized with the student denying ownership of a cheat

slip.

Element	Risk	Action	Mitigation Strategy
Student	The student denies responsibility for a cheat slip on his desk.	Transfer	Student signs an agreement that states he is responsible for any suspicious activity around his exam desk.
Grading computer	Unauthorized access to the grading computer, for example through brute force attacks. Instructor may be accused of making a change to grades he didn't actually execute.	Address	Solid and safe logging not only on the grading computer but also on the grading computer network.

6.2.4. Information Disclosure

With regard to our example, information disclosure as described in section 3.4 is manifested through students obtaining exam questions or answers before the exam. Information disclosure applies to processes, data stores and data flows.

Element	Risk	Action	Mitigation Strategy
Previous exams database	Similarity or reuse of previous exam questions makes the exam too simple.	Accept	Leave it to the instructor to modify or phase out previous exam questions.
Unwritten exam storage	The room containing unwritten exams may be suspect to breaches through various attacks, including results of human failure.	Address	Install a self-closing door that must be opened with a valid ID card from both sides. Use camera surveillance in the room.
Student desk	Proximity to other desks allows students to copy answers from other desks.	Address	Leave sufficient space between students and have assistants monitor student activity during the exam.
Copy machine → Unwritten exam storage	The assistant may accidentally drop a copy while bringing them to the exam storage.	Address	Require all exams to be transported in opaque boxes.

6.2.5. Denial of Service

Denial of service attacks are less heard of in our example environment, but may prove to be very effective. Such attacks can apply to processes, data stores and data flows and are further described in section 3.5. Much like with technical examples, these attacks are more effective when done in a distributed manner, i.e. by a group of students.

Element	Risk	Action	Mitigation Strategy
Assistant	A group of students occupies all assistants with exam related or even irrelevant	Address	By policy, one assistant must remain unoccupied and continue

Element	Risk	Action	Mitigation Strategy
	questions, during which other students can do deceitful activities like exchanging answers.		to monitor student activity.
Bathroom break	Many students request bathroom breaks at the same time, pressuring assistants into allowing more than one student outside of the room at a time.	Address	Designate a specific bathroom per exam and have extra assistants on call to monitor bathrooms that expect more than one student per exam.

6.2.6. Elevation of Privilege

Elevation of privilege entails allowing someone to do something they are not allowed to do and is further described in section 3.6. STRIDE-per-element considers elevation of privilege attacks specific to processes.

Element	Risk	Action	Mitigation Strategy
Secretary	Student may bypass security checks on grading computer act as a secretary that would like to enter grades.	Address	Put an air gap between the grading computer network and the remaining university.
Copy machine	Student bypasses the copy machine's security checks when accessing memory containing previously made copies	Accept	Low risk. Copying machine for exams is not publicly accessible and locked behind a door.

6.3. Results

Students are always looking for ways to optimize their grades, but luckily by the time they reach university many have grown out of the age where cheating is a viable option. To deter the remaining few, universities often have generations of experience on how to best monitor and catch cheating students. This often entails a certain level of trust towards the students and playing with the fear of consequences. In addition, since students only spend a limited time in their university, the risk of finding major security issues is rather small.

On the other hand, it is hard to determine how many students have successfully cheated without getting caught. While some students brag about their achievement, others keep quiet and enjoy their improved grades.

If you are reading this as a student, it is important to thoroughly analyze the security measures established by your university and concentrate on the unexpected. Of course you should always have a backup plan in case you get caught and determine if cheating is really worth the risk.

Should you be reading this as an assistant or lecturer, take a moment and apply the STRIDE principal to your own university exam process. Even if you don't end up instating security countermeasures to address the more bizarre risks, it could help keep an eye out for suspicious behavior and help convict a few more cheating students.

7. Conclusion

Thinking like a hacker is particularly hard if you are not a hacker. Aside from what you've seen in the movies and general technical experience, there is bound to be at least one more way to breach a system that you didn't think of.

Threat modeling provides you with a structured approach to finding potential security issues. Following a software-centric approach will harden your software from the inside, making it difficult for attackers to breach your systems and steal confidential information. In addition, the documentation mechanisms help avoid security regressions when modifying code.

While commonly used in software projects, this paper has shown that threat modeling can also be applied to non-technical systems, helping to find weaknesses. Even if you don't make threat modeling an integral part of your next project, having a mindset for the topic will help you build better and more secure software from the start.

Bibliography

Blunden 2013 — **Bill Blunden**. *The Rootkit Arsenal: Escape and Evasion*. Burlington, Massachusetts: Jones & Bartlett Learning, 2013.

Ciampa 2005 — **Mark Ciampa**. *Security Guide to Network Security Fundamentals*. Boston, Massachusetts: Thomson/Course Technology, 2005.

Cole 2002 — **Eric Cole**. *Hackers Beware*. Indianapolis, Indiana: New Riders, 2002.

Cornutt 2013 — **Chris Cornutt**. *Core Concepts: Trust Boundaries*. http://websec.io/2013/08/27/Core-Concepts-Trust-Boundaries.html. Zuletzt abgerufen am 30.12.2014.

Dulaney 2011 — **Emmett A. Dulaney**. *CompTIA Security+ Deluxe*. Indianapolis, Indiana: Wiley, 2011.

Engebretson 2013 — **Patrick Engebretson**, *The Basics of Hacking and Penetration Testing*. Amsterdam: Syngress, an imprint of Elsevier, 2013.

Fernandez-Buglioni 2013 — **Eduardo Fernandez-Buglioni**. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Hoboken, New Jersey: Wiley, 2013.

Graham/Olson/Howard 2011 — **James Graham, Ryan Olson, Rick Howard**. *Cyber Security Essentials*. Boca Raton, Florida: Auerbach Publications, 2011.

Hong Kong 2010 — **Hong Kong Special Administrative Region**. *Wireless Network Security*. http://www.infosec.gov.hk/english/technical/files/wireless.pdf. Zuletzt abgerufen am 30.12.2014.

Howard/LeBlanc 2002 — **Michael Howard, David Leblanc**. *Writing Secure Code*. Redmond, Washington: Microsoft Press, 2nd Edition 2002.

Jakobsson/Myers 2007 — **Markus Jakobsson, Steven Myers**. *Phishing and Countermeasures*. Hoboken, New Jersey: Wiley-Interscience, 2007.

McGraw 2006 — **Gary McGraw**. *Software Security: Building Security in*. Upper Saddle River, New Jersey: Addison-Wesley, 2006.

Michalakis/Soulé/Grimm 2007 — **Nikolaos Michalakis, Robert Soulé, Robert Grimm**. *Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks*. https://www.usenix.org/legacy/event/nsdi07/tech/full_papers/michalakis/michalakis.pdf. Zuletzt abgerufen am 30.12.2014.

Microsoft 2012 — **Microsoft SDL Process Guidance**. *Appendix N: SDL Security Bug Bar*. http://msdn.microsoft.com/en-us/library/cc307404.aspx. Zuletzt abgerufen am 30.12.2014.

Mirkovic et al 2005 — **David Dittrich, Jelena Mirkovic, Peter Reiher, Sven Dietrich**: *Internet Denial of Service: Attack and Defense Mechanims*. Upper Saddle River, New Jersey: Prentice Hall Professional Technical Reference, 2005.

Nahari/Krutz 2011	**Hadi Nahari, Ronald L. Krutz**. *Web Commerce Security*. Indianapolis, Indiana: Wiley, 2011.
Paul 2014	**Mano Paul**. *Official (ISC)² Guide to the CSSLP CBK*. Boca Raton, Florida: CRC Press, 2014.
Peng 2014	**Kun Peng**. *Anonymous Communication Networks*. Hoboken, New Jersey: Taylor and Francis, 2014.
Shostack 2014	**Adam Shostack**. *Threat Modeling: Designing for Security*. Indianapolis, Indiana: Wiley, 2014.
Westfall 2009	**Linda Westfall**. *The Certified Software Quality Engineer Handbook*. Milwaukee, Wisconsin: ASQ Quality Press, 2009.