Module 14: Security Engineering; Resilience Engineering

Objectives:

1. Know the difference between application security and infrastructure security.
2. Become familiar with some guidelines for designing secure systems.
3. Understand the concept of survivability, and know why it is important.
4. Be introduced to some additional approaches for validating and verifying a system to assure dependability.
5. Be introduced to the notion of using statistical testing to test the dependability of critical systems.
6. Understand why process assurance is important in dependability assurance.

Assigned Reading:

Sommerville, chapters 13 &14

Assignment:

No assignment for Module 14.

Quiz:

There is no quiz for Module 14.

Lecture:

The Increased Need for Security

System security has become increasingly important in recent years, particularly since the use of the internet has become so widespread. Networked systems are under a constant barrage of viruses, trojans, denial of service (DoS) attacks, phishing emails, and other types of intrusions. If you have an anti-virus application installed and working on your system (most of you probably do), and you are able to check the statistics of how many attacks have been detected and prevented, you may be surprised at how many attacks have been deflected without your even realizing they were made — assuming, of course, those data are accurate. Even more astounding is the total number of virus and worm variants, trojans, and other attacks that anti-virus programs test for. Most attacks are made with some level of malicious intent, although some attacks are made just to showcase the capabilities of the attacker. In rare cases, an attack may arise as an emergent property of a networked system. An example of an emergent type of attack would be a denial of service attack that results from a large number of concurrent requests for a particular resource, effectively slowing down a portion of a network, or even bringing that portion of the network to a complete standstill. The effects of a security breach on a system are almost always negative. Data may be corrupted or stolen; service may be interrupted for an indefinite period of time; individual computers may become so bogged down with malware that, for all intents and purposes, they cease to function.

Because we now tend to rely so heavily on software, especially networked software, for many of our needs, such as banking, shopping, and storage of proprietary data, software engineers must include system security as a vital component of a system's requirements. Sommerville distinguishes between two levels of security:

1. Application software security
2. Infrastructure security

Application security involves only the application itself. The engineers who build the application are responsible for ensuring the application itself has a capable defense. Infrastructure security covers all software on which the applications depends. This includes the operating system, component libraries, the database management system (if any) used by the application, and any other software that interacts with the application such as middleware or other applications such as web browsers. Interestingly, most attacks target the infrastructure, since the infrastructure typically supports many other applications as well, and thus its use tends to be widespread. If security holes are found in the infrastructure, the cognoscenti of the hacker world all learn about them very quickly, and begin exploiting them immediately, before patches are built to plug the holes. The responsibility for infrastructure security lies primarily with management, since application engineers usually have very little control over ensuring the defense of infrastructure software. Management must ensure that a robust security policy is in place, and they must employ whatever security measures are afforded them by the infrastructure software, as well as software developed expressly to protect systems from attacks. User authorization and authentication are both essential, since many attacks can stem from inside knowledge of the system.

Some General Design Guidelines for Ensuring Security

Sommerville lists 10 general guidelines for ensuring system security at the level of system design. The extent to which these and other guidelines are followed depends on the level of security needed. A Department of Defense system, for example, will likely require much stronger security than a public blogging system. It is confounding to me why so many systems on the internet these days requires a username and password for access, when there is no apparent need for such security. For example, the New York Times website allows free access to all content on its website, although some articles require that you have registered yourself with a username and password. There is no exchange of money involved, and no financial or geographical data are directly required; yet, if you do not register, you cannot view certain content.

In any event, those who are developing systems that do warrant some level of security may want to pay attention to the following guidelines:

1. Have an explicit security policy, and base all security design decisions on this policy.

   In essence, this guideline strives to ensure that security design decisions are all based on a common policy, and that these decisions will be addressed by the system requirements.

2. Avoid a single point of failure.

   More than one mechanism should be used to ensure the security of the system. The series of secret questions and answers now common in many banking websites, among others, is one example of this. Not all the different mechanisms that are employed need to apply at the same level of security, however. For example, in addition to employing user authentication measures, each user may be placed into one or more permission groups that only allow certain users access to certain system capabilities. Thus, a low-level user would not be able to access higher-level capabilities merely by being logged in to the system. Additionally, for example, safeguards could be employed to backup all changes to data in a log, so in the event data are corrupted by an attack, they can be restored from the backup log.

3. Ensure that when the system fails, it is still secure.

   If a secure system fails, the failed system state may be exceptional in that data from an incomplete transaction may persist when it would have been deleted had the transaction been successful. Sommerville gives a good example of how this might happen, where sensitive data may be stored temporarily on a client computer during a transaction. Normally the

temporary data are removed following the transaction, but if the transaction fails, the cleanup process may be unable to complete the removal of the data on the client. In this case, ensuring the stored data are encrypted provides a layer of security in the event of a failure.

4. Balance security with usability.

   Some of the burden of ensuring system security falls on the shoulders of the users. Users may be required to remember a username, password, and the answers to one or more secret questions. They may be required to change this information periodically. Users are almost always advised of some common techniques to ensure the security of their login information. Unfortunately, many of those techniques result in login information that is difficult to remember. In some cases, users may be required to use only specially designated terminals to access certain resources, which can cause problems if there is competition for the use of those terminals, or if they are in inconvenient locations. When usability begins to suffer as a result of security measures, users may compromise security by writing down login information, finding ways to login from unauthorized locations, etc. The bottom line here is that system security should be designed to have as minimal an impact on usability as possible. Convincing the users of the importance and necessity of the security measures being used is essential to this effort.

5. Log user actions.

   Keeping a log of all user actions serves at least two important purposes. First, in the event of a system failure or a security breach, it may be possible to use the logged information to determine who breached the security, and from where; the log may also allow restoration of any data that may have been corrupted or deleted in an attack. Second, notification that users' actions are logged may help to deter at least some individuals from attempting to breach system security. These days, now that storage space is no longer a serious limitation, there is really no good reason not to maintain user activity logs. Of course, the data in such logs become part of the system, and therefore should be protected by encryption, and by restricting access permission to only certain, high-level users.

6. Use redundancy and diversity.

   We have already discussed what redundancy and diversity are, and how they can be useful in preventing system failures. They can also help to prevent attacks and/or mitigate the impact of an attack. For example, maintaining redundant copies of sensitive data helps to ensure the data can still be made available if one or more copies of the data are corrupted. Using diverse hardware and software infrastructure can help minimize the spread of an attack. An attack that exploits a security hole in the Windows operating system will be unlike to work on a system running Linux.

7. Ensure all inputs are validated.

   A great many system failures, as well as security breaches, can be initiated by feeding a system invalid data. Very long input strings, for example, can be used to cause buffer overflows. If the input string includes characters that can be interpreted as executable code at the location of the buffer overflow, it is possible for an individual to remotely gain control of a system. It is also conceivable to cause a denial of service attack by having an automated bot fill a database with bogus records to the point that all available hard drive space is used, which in turn will cause the system to bog down or even stop executing. A similar effect can be caused by repeatedly (and very rapidly) generating large numbers of errors, causing log files to swell, thus reducing available storage space. Those areas of a system that are directly associated with receiving input, whether from human users or from other systems, should be designed to validate all input data before allowing it to be processed. Validation should include not only the actual content of the input data, but also the frequency with which inputs are made. Since another software application can be set up to feed data into a system much more rapidly than a human could ever enter the data manually, a sudden burst of rapid inputs may signal that an attack is underway. This type of protection is used in part to prevent the rapid proliferation of spam, which may contain infectious attachments, phishing requests, or which may be sent simply to try to bring down an email server.

8. Compartmentalize assets.

   I alluded to this in a previous paragraph. The basic tenet of this guideline is to ensure that users are grouped in terms of the types of access they should be allowed to have, rather than allowing full system functionality to be available to everyone. A temp worker hired to enter data into the system should not be allowed to edit or delete any records, since they could either accidentally or intentionally misuse their system access to modify or delete records that should remain untouched. A permanent employee with a higher permission level may be allowed to edit or delete data pertaining to certain fields of a record, but not the entire record itself. Someone with still higher access privileges may be required to delete an entire record. Protected assets can be compartmentalized in any number of ways, although it is possible for conflicts to occur. For example, a user may need to be placed in a particular group in order to have access to a certain feature, but being made a member of that group may afford that user access to other features they should be restricted from using. It can be difficult, and quite complicated, to arrange an ideal compartmentalization scheme for a complex system.

9. Design for deployment.

   It may seem superfluous to mention this as a guideline, but a system should be designed with the expectation that it will be deployed (as

though any engineer would set out to design a system that is never expected to be deployed). The main point here is that many of a system's security features are configurable, in order to allow customized protection depending on user and organizational needs. If the system's security is to be properly configured, it must be relatively easy for the individuals who install the system to configure the security settings properly. If the desired security level changes over time, it should be easy for authorized users to adjust the security configuration as needed. This can be accomplished in large part to having an intuitive user interface that allows visualization of all security settings along with the capability to change them. Good documentation that details precisely what each security setting does, and guidelines for setting the security configuration, is also essential.

10. Design for recoverability.

    I've also alluded to recoverability in several places so far. In the case of security, recoverability includes not only system failures, but also security breaches. If a security breach is detected, there should be provisions for limiting access to the system until the stored data can be validated, with any missing or corrupt data restored from backups and/or log files. If someone manages to compromise an encryption scheme, all data must be re-encrypted using a new scheme. It may be necessary to require all users to change their login credentials. During the recovery process, it may be possible to identify the attacker based on data collected in log files, such as IP addresses, or which login credentials were used to initiate an attack.

Survivability, or Resilience, of a System

If one had a completely standalone application, that ran on an operating system that was developed by the same organization that built the application, and all software with which the application must interact is also developed by the same organization—that is, a completely in-house system, the likelihood of a successful attack penetrating the system might be relatively low. The reasoning would be that because every component of the system is known, successful attacks can be minimized. There would still be some possibility that sooner or later an attack would get through, because no system is perfect, and as we've seen, software changes over time. When you consider the fact that most systems in use today are not isolated, as the hypothetical system described above, it is reasonable to assume that the system will succumb to multiple attacks throughout its deployment.

For a critical system, we would like for the system to still be able to provide as many essential services as possible to its users, even after an attack has penetrated the system. The capacity of a system to continue in this way is known as **survivability**, or **resilience**. Engineering survivability into a system

first requires that the most essential system services be identified, with the goal that those services at a minimum will be able to continue despite a successful attack. Next, the vulnerable points of the system need to be identified. Once the vulnerabilities are known, resistance mechanisms can be focused primarily on those areas. The system also needs to be capable of detecting when an attack has occurred, and what services have been compromised. Finally, the system must attempt to continue to provide essential services until the attack can be neutralized, possibly by using redundant subsystems and data backups. After the attack has been neutralized, the security hole that allowed the attack to succeed can be fixed, and functionality using the primary systems can be restored.

The High Cost of Verification & Validation

Ensuring the dependability and security of a system, particularly a critical system, requires a great deal of additional effort during the validation and verification (i.e., testing) phases. Greater effort is also required for the other phases of development, but ultimately, testing will determine whether or not the system is likely to live up to expectations. The number of test cases that are run will be much higher than for non-critical systems, and more types of testing will be performed. Some verification will not even involve the execution of any code. The additional cost involved in such thorough testing is prohibitive for pretty much all but critical systems, but when a system failure could result in personal injury, death, environmental damage, or significant loss of money, the extra time and cost are worth it.

Enhancements to Standard Testing Procedures

Typical software testing consists of a mixture of black box and white box testing. However many test cases are used for a non-critical system, you can be sure there will be more test cases for a critical system. More paths will be tested, more input combinations will be tested—all to help provide confidence in the quality of the system. In addition to the usual testing procedures, such as branch testing and basis path testing, critical systems will be subjected to such procedures as stress testing, where the system is run at full capacity or above full capacity, since the system may become overloaded at certain times during execution. For example, an online banking system may need to handle a very large number of transactions during certain peak operating hours. Some types of attack also rely on overloading a system, to take advantage of its reduced capacity.

Code Reviews

Before source code is ever executed, it may be subjected to review by other software engineers. During a code review, a group of engineers will analyze the code by examining it for instances of poor programming practice, places where additional efficiency may be introduced, places that are likely to lead to system vulnerabilities, etc. The code may also be manually traced to identify any glaring faults. The theory behind code reviews is that the time spent simply reading through the code will identify a large number of obvious faults that can be corrected before the code is actually tested. That means there will be fewer faults that must be fixed during regular testing, with less subsequent regression testing required not only to verify that the original fault was corrected, but also that no new faults were introduced.

This type of code analysis can also be automated, to an extent. Automated code checkers can identify certain patterns (e.g., infinite loops, potentially exponential recursion, unused variables, unreachable code, etc.) that are either faulty, or that are likely to cause problems when the software is in operation. Automated checkers are not foolproof, however, so even if automated checkers are used, it will still be useful to have the eyes and minds of other, human, software engineers analyze the code. Some IDE's have limited code analysis capabilities that can draw attention to the more common problems such as unreachable code, unused variables, and variables that are declared, but not initiated.

Formal Verification

In some cases, it may be possible to employ mathematical proofs to show that a particular section of code, or even an entire program, meets its specification. If the specification is assumed to be correct, then logically the code must be correct. Formal verification is usually only used in those cases where formal methods can be used during requirements gathering and/or design. In those situations, where you have a specification that has been defined using formal methods, it is possible to use theorem provers to show whether or not a specification is correct. Testing a specification in this way is known as **model checking**. A model-checking language is used to construct a model of the system, with formally defined properties that should hold true for a correctly operating system. The model checker analyzes all possible state transition combinations, and tests the truth values for the properties of interest. If a property does not hold true for even a single state in the model, the specification is incorrect.

System models can be created from source code, as well as from a specification. This method is sometimes preferable, since the formal structure of source code permits models to be generated automatically. With a specification that is not defined using formal methods, models must be constructed manually, which is not only time consuming, but also error prone.

Statistical Testing

Some critical systems are so vital, it is desirable that they virtually never fail. One definition, which may be somewhat outdated now, states that a critical system should not fail more than once for every 10^9 hours of continuous operation—that's approximately 114,000 years! Two things are notable about that requirement:

1. There is no way to actually run a system for 114,000 years to see how many times it fails in that time. Humans only live to be around 75-80 years, on average, and even if we lived long enough, it's doubtful the hardware used by the software system would survive.
2. It isn't practical to expect software to survive this long anyway, since it would undergo innumerable changes along the way, and with each new version the time clock would start over at 0.

So why use such an impossible figure? The answer is that even though a system cannot be physically tested for such a long time span, there are ways to test the system for a shorter period of time and use statistics to show that if indeed the software were to run continuously for 10^9 hours, it would fail no more than one time. Statistical testing can be extremely costly. It is necessary to generate an operational profile of the system; i.e., a profile of how the system will be used once it has been deployed. The biggest problem here is that it is hard to know how accurately the generated profile conforms to reality. Once the profile has been created, a large set of test data must be generated that fits the operational profile. Generating all this data can be very time-consuming and costly, and the accuracy of the data depend entirely on the accuracy of the operational profile. After the test data have been generated, the system is tested using those data, and the number, type, and time of each failure is recorded. In some cases, multiple copies of the system may be run in parallel to extend the projected testing time. For example, 10 copies of the system operated for 1000 continuous hours are assumed to represent a single copy of the system operated for 10,000 continuous hours. This testing process must be run until a statistically significant number of failures have been recorded. Unfortunately, there is no way to guarantee how long it will take for a statistically significant number of failures to occur, and until they do no statistically reliable conclusion can be drawn about the system.