ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 34 — Security

Jeff Zarnett & Mahesh Tripunitara

Security

It would be remiss of us not to talk a little bit about embedded systems security. Security is a very big concern and a few years ago it was really a hot topic. Embedded systems have a number of challenges in security that you should be aware of before leaping into developing for them in the real world.

Introducing this topic so late in this term violates one of the most important rules about security: you can't "bolt on" security to your system after you've created it. It should be designed with security in mind from the beginning. Hopefully you will overlook this.

Just as a matter of terminology: an *attacker* is a malicious user who is trying to damage or exploit the system. A *countermeasure* is some action or process we can take or implement to defend against attacks. An attacker is successful if he or she gains access to information he/she shouldn't have, damages the system in some way, or interferes with its legitimate operations.

Errors in the software design and implementation are responsible for many security challenges, whether in embedded systems or not. Attackers might give carefully crafted input to the software to trigger a crash or gain access to administrative privileges. There are, naturally, many software development techniques that are beneficial in defeating attackers. As you may have guessed, there is a 4th year course on the subject of security, and the details on software security belong to that course. However, we will discuss the intersection of security and embedded systems.

Properties of Embedded Systems

There are a few notable properties of embedded systems that have security implications.

Physical Access Probably the most critical issue is physical access: a great number of security measures are easily defeated if an attacker has physical access to the device. Many embedded systems are out there in the world and not kept under meaningful physical security. ATMs (Bank machines) are often easily accessible; contrast this against the bank's servers, which are (hopefully) kept under strict security lockdown at the bank's data centres.

Limited Resources While a typical desktop has the power and processor cycles necessary to run security checks, in embedded systems the situation is different. Virus scanners make sense on a desktop, but an embedded system may not be capable of that. Only limited resources are available to use for security [PW08].

Hard to Update It might be difficult or impossible to update an embedded system, for many reasons. If you discover a security vulnerability, you can't just create an update and expect it to get pushed out in Microsoft's next "Patch Tuesday".

Connectivity A remarkable number of embedded systems are connected, either to the internet or to other systems. This provides a gateway for attack. If a system is connected to the internet, then it can be exploited from anywhere in the world [PW08].

Attacks on Embedded Systems

How might an attacker go after an embedded system with the goal of causing damage or getting access to data they shouldn't have? There are virtually unlimited answers to this question. We'll go over a few categories that describe different types of attacks, but remember that this list is not exhaustive. Use your imagination and you will surely come up with more.

Physical Tampering An attacker who has the physical device might be able to open it up and read data from its storage, attach listening devices to the input or output, or they could just pick it up and take it home.

In fact, the device might already be in the attacker's home! Imagine you are designing software for a cable tuner / PVR (personal video recorder) box. There are access restrictions you need to enforce, such as customers should not be able to watch channels they have not paid for. The key problem is that the box is located in the customer's living room. How do you stop the customer from tampering with it?

Closely related to the idea of physical tampering, in this kind of attack, the attacker will just take (or make a copy of) the stored data. Then they have all the time they need to work with the data.

As another example: attackers might get credit card data by stealing the credit card swipe terminals from a store. Those machines typically store information temporarily and are supposed to be cleared on a regular basis, but not every store owner follows this procedure. Once at home the attacker can read all the information stored in the terminal. He or she might then return the terminal at a later time so its presence is not missed.

Sometimes an embedded system might be disabled by simply pulling the plug.

Denial-of-Service Attack Perhaps you are familiar with the term already from websites. In short, in a Denial of Service (DoS) attack, the target system is flooded with so many requests (legitimate or otherwise) that completely overwhelm the ability of the target to respond. The problem is more acute in embedded systems because their ability to respond is limited by the resources available. A variant on this is the Distributed Denial of Service attack (DDoS) where many computers launch a DoS attack at the same time against the target.

One unusual form a DoS attack might take in embedded systems is called an exhaustion attack; the attacker causes the system to run its battery down faster (through computation, use of sensors/actuators, or disabling sleep mode). When the battery is exhausted, the system will shut down [PW08].

Sensor Tampering Attackers might be able to bypass or disable a system by damaging or disabling the sensors. Methods might be high tech (feeding in false input) or low tech (smashing a camera). Example: ever seen a spy movie where they loop a segment of the security video over and over again so the spy can go about his (deadly) business while the security guards don't notice?

Countermeasures

Fortunately, we don't have to just suffer attacks helplessly; there are some things we can do to defend against attacks. Physical security is high on this list (locking a credit card terminal to the cash register, as an example) goes a long way towards mitigating the problems we saw above, but that's not our focus. The more relevant question is, what can we do in light of the fact that physical security is an issue?

Encryption As a general note, it's good practice to encrypt all sensitive data. It's doubly important in embedded systems because they are physically vulnerable. A short definition of encryption is that it is a way of encoding data in a way that an attacker cannot read the data, but authorized parties can. Taking data and encoding it is called encryption and transforming the encoded data back to its original form is called decryption.

If the credit card swipe terminals has all its data encrypted, it means an attacker who steals the terminal will not be able to use the credit card data (easily, at least). Encryption might also be used on the system software. If you have a PVR box that you do not want users to alter the code on, you might use an encrypted software image, decrypted on boot; if the image is not encrypted with the correct key, it will not run at all.

A small note about encryption: encryption is really hard. There are a number of well-known and well-understood cryptographic algorithms. You must use one of them; if you attempt to create your own it will be vulnerable to cracking. We'll take a look at cryptography in more detail shortly.

Defeating Denial of Service Attacks DoS attacks and their defence is a relatively large area of research, and it may not be possible to defend against all kinds of attacks, particularly if it is a DDoS attack. Under those circumstances your only option might be to fail gracefully. Assuming, however, that the wrath of a DDoS attack is not called down upon your system, there are some ways to deal with a user (or small group of users) who are deliberately taking up more than their fair share of the system's resources.

One option is to limit the rate at which requests may be entered. This is commonly seen in messaging software, where there is some limit for how many messages can be sent in a short period of time. If that is exceeded, no further messages can be sent until after some waiting period. Another option is to assign the requests very low priority; the user who is taking up too much of the system's resources then has to wait while other users' requests take priority. Finally, it might be an idea to lock out the offending user (temporarily or permanently) so others' use of the system does not suffer.

Monitoring Your embedded system might have some secondary software processes to check on the health and status of the system and its sensors/actuators. This could be another thread in your process, another process running on the same system, or another embedded system that runs in conjunction with the primary system. The monitor checks the state of various parts of the primary system. It might run periodically or continuously. If something unusual is detected, the monitor software could take corrective action, trigger an alert, shut the system down, et cetera.

Cryptography

We mentioned the countermeasure of cryptography, and let us take a look now at a common encryption scheme: RSA. The scheme is named after its inventors: Ron Rivest, Adi Shamir, and Leonard Adleman. In the RSA scheme, a user, say Alice, is associated with the triple $\langle e, d, n\rangle$ with the following properties:

- Each of e, d, n is a bit-string that is interpreted as an integer.
- For any bit-string m, $(m^e)^d \equiv m \pmod{n}$.
- Given $m^e \pmod{n}$, it is difficult to discover m.
- From knowledge of only $\langle e, n\rangle$, it is difficult to discover d.

Alice publicizes $\langle e, n\rangle$ and keeps d secret to herself. Suppose Bob wants to send Alice a message in a manner that it is confidential to her. How can be use Alice's pair $\langle e, n\rangle$ to do so?

An option is, if Bob wants to send a message m, he can simply compute $m^e \pmod{n}$ and send that over a public channel to Alice. The question is, whether we should consider this to be a "good" approach. That is, does this achieve what we think of as confidentiality.

A similar question arises in the use of RSA as a signature scheme. A signature is used by a sender to indicate that a message is authentic, where authenticity is a property that is inferred from context. Assume that the RSA scheme has the following additional properties:

- For any m, it is difficult to generate $m^d \pmod{n}$ without knowledge of d.
- $\left(m^d\right)^e \equiv m \pmod{n}$.

Use $S(m)$ to denote Alice's signature on message m. Is $S(m) = m^d \pmod{n}$ a good signature scheme for Alice to use?

The answer to both these questions is generally thought to be "no." For the encryption, a problem is that $E(m) = m^e \pmod{n}$ is a function. That is, given the same m, it produces the same output. Therefore, if Alice is sent the same message twice, a malicious eavesdropper can detect that that happened. This can be seen as a violation of confidentiality.

For the signature, RSA has some additional properties that are highly undesirable from the standpoint of the above $S(m)$ serving as a signature scheme. Specifically, suppose Alice signs and sends out signatures on two messages m_1 and m_2. Then, a passive eavesdropper can exploit the property that $S(m_1) \cdot S(m_2) = S(m_1 \cdot m_2)$, where "$\cdot$" is arithmetic multiplication.

Thus, even though Alice never signed the message $m_1 \cdot m_2$, the attacker is able to generate her legitimate signature for that message. This is an example of *existential forgery*, which is a somewhat weak form of signature-forgery, but nonetheless of concern.

This same issue underlies the use of Electronic Code Book (ECB) mode of encryption with a symmetric key based cryptographic scheme such as AES. AES, like RSA, is a block cipher. That is, AES requires that we encrypt exactly 128 bits at a time. In ECB mode, given a long message that we want to encrypt, we simply break that message up into 128-bit pieces, and encrypt each in turn. This suffers from the same indistinguishability problem that RSA does, as we discuss above. Other modes, such as Cipher Block Chaining (CBC) mode, mitigate this problem. (See the Wikipedia page on modes of operation for pictures that clearly explain how ECB and CBC work.)

Solution The solution, for encryption, is to somehow randomize every instance of encryption. This is exactly what CBC mode does for AES. For every message m that we want to encrypt, we generate a random Initialization Vector (IV). This IV is xor-ed with the first 128-bit block before it is encrypted using AES. The encryption of the first block is used as the IV for the second block.

A question that arises then is: how is the IV communicated to the receiver so she can meaningfully decrypt the message? It turns out that communicating the IV even in plaintext is okay.

For the signature scheme, an option is to subject every message to a random function (a so-called hash function) before exponentiating it with $d \pmod{n}$. Such functions usually do not demonstrate the kinds of properties that can undermine a signature scheme.

Password Salting The previous section asserted that communicating the Initialization Vector in plaintext is okay. Why? Let's consider a slightly different example: passwords. In old UNIX systems, encrypted passwords were stored in a file `/etc/passwd`. Imagine you opened this file and you saw the following:

```
donna:25hg57A\%53H
edgar:96ugabfAWo95
frank:25hg57A\%53H
```

It is immediately obvious from even the encrypted passwords that users Donna and Frank have the same password. So if Frank is the one looking at this file, given that he knows his password, he knows Donna's password and can log in under her account.

If we apply the Initialization Vector technique here, what we get is a password "salt". This is some randomly generated data that we append to the user's password before running it through the encryption function. So if Donna's password is the ever-so-secure "password" and the random salt generated for her is "8z80ppEth$2", the input to the password encryption function is "password8z80ppEth$2". The salt (random data) is stored in the `/etc/passwd` file alongside the username and encrypted data and appended to the password entered whenever a user tries to log in. In this case, even though Frank might have chosen the same password, because his salt is different, he has no way of knowing that, even if he looks in the enhanced `/etc/passwd` file:

```
donna:8z80ppEth$2:YPlk63KifQ
edgar:6fPuXqdRdI:2h9u7e4K03
frank:VYlp7Whjth:Rxt!d7u6k9
```

Authentication vs. Authorization

These are two related concepts and similar words, so let us take a moment to define them formally.

Authentication verifies who the user is. In many computer systems this is done with a username and password, but it could also be done in other ways: fingerprint scan, key card, et cetera.

Authorization verifies what the user is allowed to do. It is common that users are not shown options for which they do not have permissions, but in some cases a simple message denying the action is returned (e.g., deleting a file might result in a popup that says this action is not allowed).

Both are typically seen in computer systems (and other parts of life). We will take Learn as an example. The first step in Learn is to log in - to authenticate yourself - using your UW user ID and your password. If you made a mistake in typing your password, access is denied; you are not authenticated as a user of the system. Then you go to look at ECE 155. Learn checks that you are authorized to see the class ECE 155 and grants access. There are other courses offered in a given term, but if you do not have access to them, they are not shown on the page.

Access Control

As in the real world, in software systems, we are very much interested in who has access to different things. We need not spend any time going over motivations for why there are access controls, so we can jump into some basics of how it is implemented. Operating systems control access to files as a part of the file system.

In operating systems like MS-DOS, the classic (pre-OSX) Mac OS, and Windows 95/98, files do not have permissions associated with them. It was possible to set a flag "read only" on a file, but anyone could clear that flag and delete or modify the file if they chose. Leaving aside the no-security model of Windows 95/98, there are three basic strategies we will discuss: UNIX-style permissions, Access Control Lists (ACLs), and Role-Based Access Control (RBAC). Permission settings may depend on the file system used where the data is stored.

UNIX-Style Permissions. UNIX-Style permissions are commonly used still today in a lot of UNIX and UNIX-like systems. Each file has an owner and a group, and a set of permissions that can be assigned for the owner, the group, and for everyone. There are three basic permissions: read, write, and execute (run as a program). The permissions are represented using 10 bits, where a 1 indicates true and a 0 indicates false. The first bit is the directory bit and indicates if the file being examined is actually a directory. The next three bits are the read, write, and execute bits for the owner, followed by the read, write, and execute bits for the group, and finally the read, write, and execute bits for everyone.

Effective permissions are determined by the user: the owner of the file gets the owner permissions even if different permissions are assigned to the group or everyone. Precedence goes from left to right: owner takes precedence over the group; group takes precedence over the permissions for everyone.

The permissions can be shown to the screen in a human-readable format which is ten characters long. The order is always the same, and so a dash (`-`) appears if a bit is zero (permission does not exist). The character `d` is used to indicate a directory, `r` to indicate read access, `w` to indicate write access, and `x` to indicate execute access.

Example: permissions of `-rwxr-----` indicate that a file is not a directory; the owner can read, write, and execute; other members of the group can read it only, and everyone else has no access to the file (cannot read, write, or execute).

Permissions can also be written in octal (base 8) where r = 4, w = 2, and x = 1. To get the octal representation, start with 0, and then add the value of the permissions that are present, using zero where permissions are absent. You might then have a permission that reads `750` - meaning that the owner has read, write and execute access (0 + 4 + 2 + 1 = 7); group members have read and execute access (0 + 4 + 0 + 1 = 5); everyone else has no access to it at all.

There are some more details like what the permissions mean on directories, and some advanced topics like `setuid`, `setgid`, and “sticky bit”, but we will not cover them in this course.

The obvious shortcoming of this approach is that it is very coarse-grained: there are only three groups for whom we can specify permissions. Within Linux and other similar systems there is a trend now towards using SELinux (Security Enhanced Linux) which is an Access Control List system.

Access Control Lists. In SELinux and NTFS (Windows NT File System), files are protected by Access Control Lists. In such a system, each file can have as many security descriptors as we like, in which we specify the permissions a specific user or group is supposed to have. Similarly, the list of permissions need not be restricted to read-write-execute; NTFS, for example, has several different permissions such as “Take Ownership”.

In NTFS, you can see the security descriptors for a file rather easily by right clicking it in Explorer, opening the properties dialog, and choosing the security tab. The descriptors have two checkboxes: allow and deny, with deny taking precedence over allow.

Permissions can start out at “default deny”, in which case only the users explicitly granted access may access the file, or “default permit” in which case everyone has access, minus those users whose privileges are explicitly denied. Default deny is obviously more secure.

A simple way to represent an access control list is a set of tuples listing the users and permissions. For example:

```
(alice, read)
(bob, read)
(charlie, read)
(charlie, write)
(bob, execute)
```

Access Control Lists have a complexity that the UNIX file permissions do not: the idea of inheritance. We’ll examine, briefly, inheritance in NTFS. Inheritance is supposed to be a convenience feature, but it can often be a cause of problems. If there is a directory with ACLs established and object inheritance is enabled, then any new file crated in this directory will receive the same ACL of the directory. The danger is that any existing file moved into the directory will retain its original ACL. If it is supposed to get the ACL of the directory, it needs to be explicitly set [Dew04].

Role-Based Access Control. An extension of access control lists is the idea of Role-Based Access Control (RBAC). In RBAC, a set of roles is created, users are assigned roles, and access is granted or denied based on the role(s) a user has or lacks. Thus, assigning rights to users is done by assigning roles to users. A user can have more than one role, but must have at least one.

Example: only members of the accounting department may read the payroll information. Thus, the payroll documents are marked as being accessible only by accounting. When a user tries to access a payroll document, the user's roles are checked. If the user's roles contains accounting, access is granted; otherwise access is denied.

An advantage this has over ACLs is that assignment is simpler; if there are many files which can only be accessed by system administrators, it is much easier to assign a new employee the system administrator role than it is to go to each file and add that employee to the file's ACL.

Like ACLs, we can write out the permissions simply as a list of tuples, this time of roles and permissions rather than users:

```
(Doctors, read)
(Nurses, read)
(Technicians, read)
(Technicians, write)
(Technicians, delete)
```

There can also be relations between roles: doctors might be able to do all the things a nurse can do. Rather than have extra permissions assigned everywhere, the system can be set up so the doctor role *subsumes* the nurse role: the doctor role has all the rights of the nurse role, and may have others.

(Author's note: my Masters Thesis is on the subject of Role-Based Access Control in Java [Zar10])

Viruses, Worms, and Trojans, Oh My!

Though this brief introduction will not replace a security course, it is important for us to go over a few of the different kinds of malicious software (sometimes called *malware*) so you are familiar with how they are classified. The definitions below are from [Sys13] and examples from [Str08].

Viruses. Before the advent of the internet, the virus was the most common kind of malware. It propagates by inserting itself into and becoming part of another program, much like a biological virus enters cells of the infected person and uses the cell to replicate. Most viruses are attached to an executable file, meaning they do not run until the user starts the program. A virus might be annoying, or it might damage some data or software. Viruses often spread through sharing: of software ("check out this cool little game"), documents (MS Word or otherwise), or, in the old days, floppy disks.

As an example, in 1999 the world faced the "Melissa Virus". It was spread by Microsoft Word Document, replicating itself and sending out copies to 50 people from the victim's address book. The virus could only activate when the user opened the infected Word document. Due to the flood of e-mail traffic, some companies suspended their e-mail service until it was resolved.

Worms. Worms are like viruses in that they spread copies and can do damage, but unlike a virus, the worm needs no host: it is a standalone piece of software. They often exploit a vulnerability of the system and take advantage of the capabilities of that system to spread.

In 2001, the Code Red worm used an operating system vulnerability in Windows 2000 and Windows NT. Other than try to spread itself, it would deface web sites and try to send many requests to the White House website, overloading their web servers. Microsoft released a patch that fixed the vulnerability, but didn't automatically remove the worm from infected machines; users had to do it themselves.

Trojans. A Trojan is another type of malware, named after the wooden horse the Greeks used to sneak into the city of Troy[1]. Users are tricked into installing it in their system, and usually gives access to the attacker who can then control the system. Unlike viruses and worms, Trojans do not attach to files or self-replicate. It usually consists of a client and server part. The victim unwittingly installs the server on his/her computer and the attacker controls it remotely with the client.

There are few famous examples of Trojans the way that we find them easily for viruses and worms, because these pieces of software do not replicate and rarely attract the attention of the media. From about the year 1998, however, there was a fairly popular Trojan called "Back Orifice". It consisted of a simple client and server and an attacker could execute the following operations on the victim's machine [Sym10]:

- Execute any application on the target machine.
- Log keystrokes from the target machine.
- Restart the target machine.
- Lockup the target machine.
- View the contents of any file on the target machine.
- Transfer files to and from the target machine.
- Display the screen saver password of the current user of the target machine.

References

[Dew04] Brian Dewey. Understanding ACLs in NTFS, 2004. Online; accessed 13-December-2013.

[PW08] Sri Parameswaran and Tilman Wolf. Embedded systems security – an overview. In *Design Automation for Embedded Systems*, 2008.

[Str08] Jonathan Strickland. 10 Worst Computer Viruses of All Time, 2008. Online; accessed 23-December-2013.

[Sym10] Symantec. Information on Back Orifice and NetBus, 2010. Online; accessed 24-December-2013.

[Sys13] Cisco Systems. What is the difference: Viruses, worms, trojans, and bots?, 2013. Online; accessed 23-December-2013.

[Zar10] Jeffrey Zarnett. Method-Specific Access Control in Java via Proxy Objects using Annotations. 2010.

[1]History refresher: Greece and Troy were enemies and at war. The Greek army was unable to get past the walls of the city of Troy. So they had a clever plan: they built a giant wooden horse and hid soldiers inside. They tricked the Trojans into bringing the horse inside the city. At night, Greek soldiers came out of the horse, opened the gates, and let the rest of their army in!