Lecture Notes

Chapter 5:
Embedded Security

CYENG 351: Embedded Secure Networking

Instructor: Dr. Shayan (Sean) Taheri

Gannon University (GU)

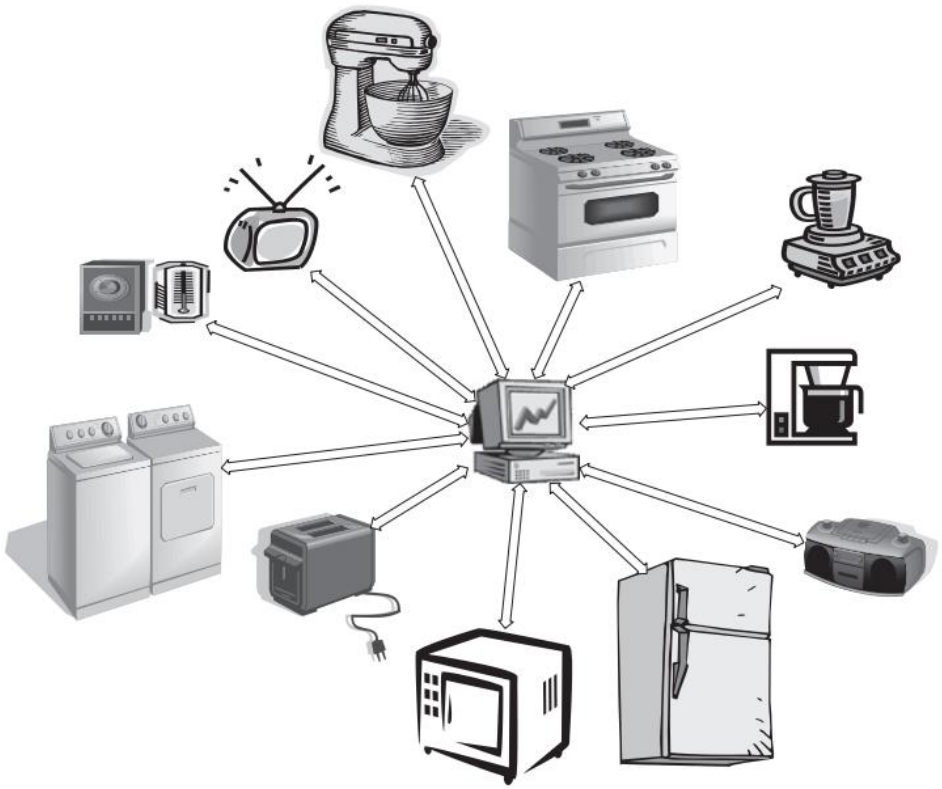
Chapter 5 Overview

- In this chapter, we will cover **security from an embedded systems developer's point of view**.
- The embedded systems we are covering are NOT PCs or expensive devices running Windows or Linux.
- The personal computer and many expensive embedded devices have **virtually infinite resources** when compared to low-power, inexpensive, embedded platforms.
- The problem is that many security solutions rely on this almost limitless supply of memory and clock cycles.
- **Cryptography** is expensive, and the flexibility of some protocols (for example, allowing multiple algorithms to be used at any give time, such as SSL does) chews up a lot of code space and data memory.
- To further exacerbate the problem, embedded systems often require better response times than PC's, seeing as they are often in control of physical systems.
- Some security solutions that work well in resource-rich environments may simply not fit the criteria for many applications due to memory or clock cycle usage.
- For embedded machines, we need to design security based upon the target platform and application, as a general security solution will often be impractical or impossible to use.
- As in the PC world, the security solution applied to any system depends on the application.
- For example, a web browser used for online shopping has much different requirements than a network-accessible payroll database.
- In **the embedded world**, the application matters even more, since there are very tangible tradeoffs in cost, performance, and security (along with area/space and power consumption).
- In both PCs and embedded systems, the application matters, but for security on an embedded device, the target platform matters as well.

Chapter 5 Overview (Cont.)

- A programmer in the PC world can treat all systems essentially the same (we handle discrepancies in platform performance with the "**minimum requirements**" disclaimer on the application's box).
- For embedded applications, the developers cost, size, and performance of the platform all matter.
- In many cases, the budget of a project may limit the hardware that can be used.
- This does not mean we should abandon all hope of developing secure applications, but it does mean we need to be clever in choosing what mechanisms and how we used them to secure our "**lean-and-mea**n" applications.
- We will look at the tradeoffs of cost and performance when implementing different security options.
- Why not just go online and find a security package for your hardware and just buy and install it? → The truth is, there are no general-purpose security packages available for a large number of embedded platforms.
- Furthermore, there are no embedded security standards in existence.
- This obviously leads to a dearth of off-the-shelf security packages for most embedded devices.

Chapter 5 Overview (Cont.)



Networked Home Appliances

(from **Networked Embedded Systems and Resource Constraints**)

PC vs. Microcontroller

Networked Embedded Systems and Resource Constraints

- As the Internet grows, an increasing number of embedded systems are going online.
- From cell phones with web browsers and networked refrigerators to monitoring pipelines and factory floor remote control, the age of the network for embedded systems is upon us.
- It used to be that all the networking technology was reserved for expensive hardware and big corporations.
- The explosion of the "**personal Internet**" has brought the technology down in price to the point where embedded devices can utilize it effectively in new, novel applications that we could not even dream of fifteen years ago.
- **Wireless technologies** have created new challenges for security, since we are creating ever smaller and more mobile networked applications.
- As the concept of "**Personal Area Networks**" (**PANS**) becomes more and more of a reality, the need for security will be greater than ever, since everyone will essentially be broadcasting information to the world.
- New technologies like **Bluetooth** and **ZigBee** allow for the creation of endless numbers of networked devices.
- With all that information flying around, we are headed for interesting times, and security will need to figure prominently in this new reality.
- The Internet has been restricted to PCs and large, relatively expensive hardware.
- As technology improves, we are starting to see some of the Internet technologies filter down to less expensive, smaller hardware.

Networked Embedded Systems and Resource Constraints (Cont.)

- We can now realistically entertain the idea of massive swarms of tiny machines that self-organize into large ad hoc sensor networks, or complex control systems that require thousands of devices to intercommunicate.
- These applications require **low-cost hardware** to be economically feasible, and they need to be small.
- These types of applications will only work if the hardware being used is small enough for the application.
- A sensor network of millions of devices is not very useful (not to mention prohibitively expensive) if every device is the size of a baseball.
- For these types of applications to work, we need to look at **limited-resource systems**.
- How do we use the scant resources on these devices to implement our applications?
- These applications will almost definitely require some type of communications security, since they will need to communicate with one another. → How do we fit the application and security into a device that likely has at most a couple hundred kilobytes of space for both data and code?
- **The miniaturization of Internet hardware** also has an interesting side effect in the embedded industry in general.
- Miniaturization costs a lot of money; as a result, many technologies are going to be kept closed and proprietary. → Given that, how do we reconcile the utility of embedded applications with the need for security?

Networked Embedded Systems and Resource Constraints (Cont.)

- We need to look at **how security applies to these applications** and how it can be implemented.
- **Implementing security takes resources, and for these applications, cost is going to be an issue**.
- If you could reduce the size of onboard memory on any embedded device, you could likely reduce cost since memory size and price are typically related (except in some cases usually with obscure, obsolete, or discontinued parts).
- If you could save just one dollar per device, it could save a lot of money if you were making hundreds or thousands of devices.
- Even with a small number of devices you might be able to reduce the size and cost of each one.
- Since professional developers continually have to balance costs with sound engineering (which are often at odds), security becomes a point of contention.
- **Security is usually big and slow, and therefore expensive**.
- The problem is that the lack of security can also be expensive, but in less clear ways.
- Many developers may be tempted to leave out security entirely in order to keep costs down and release products.
  - It is even more likely that the engineer will not be part of the decision at all, a manager makes the decision and the engineer can only voice unheard complaints.
  - But what if that engineer could show that the addition of security would not cost too much, since the security necessary is designed into the system and only uses a small portion of the available resources?

Networked Embedded Systems and Resource Constraints (Cont.)

- This would almost completely eliminate the conundrum of whether an application should be secure—all applications can be secure.
- Engineers are happy with the sound engineering choice, and management is happy since they can pay a small amount to have the confidence that the application will be relatively secure. → But just **how do we make a constrained system secure**?
- Most security protocols and mechanisms are designed for systems with nearly limitless resources (at least as compared to our small embedded devices).
- They require **fast processing** and **lots of memory** (**both for code and data**).
- On top of that, security makes life difficult to design applications that need to react to real-world inputs.
- **Many applications just do not have the time to wait for a generic security protocol to finish a communication before an action is required in response to some input**.
- With the limited resources of our embedded systems, we have a serious barrier to implementing adequate security to our applications.
- **One way to overcome the barrier of limited resources for security is to tailor the application to work with the security**.
- There are obviously other ways to overcome the barrier, such as adding more resources, but we want to keep the price of the hardware down, so it is usually better to design the software to fit the desired hardware than the other way around.
- **Our Interest**: Analyzing security protocols so that we may tailor our applications to the security desired, thereby freeing up resources and reducing the overall cost of the hardware required for those applications.

Embedded Security Design

- The basic idea for tailoring applications to utilize security mechanisms and protocols effectively is to start off with a design that takes security into consideration from the start of requirements gathering.
- Without starting the application with security in mind, the only option is to try and shoehorn in a security solution later on, which leads to additional headaches, schedule slips, and worst of all it can compromise the security of the application because the interaction between **the bolt-on solution** and the application may not be well defined.
- To begin the design of a secure embedded application, we need to look at **all of the tradeoffs between performance, cost, and security (along with area/space and power consumption)**. → Unfortunately, these three concepts are almost always directly at odds with one another. → More performance means the cost goes up, lowering the cost means lowering security and performance, and implementing higher security means performance will decrease.
- With all these tradeoffs, where do we begin? → Well, **the best place to start is at the beginning**—what is the application going to do, and how is it going to accomplish that goal?
- As an example, let's compare the security requirements for a couple of "**big iron**" (i.e., nonembedded with lots of resources) applications—**a web store** and **a payroll system**.
  - Obviously, these two applications have drastically different security needs (it should be obvious that both should be secure applications).

Embedded Security Design (Cont.)

- The web store expects interaction between its servers and hundreds or thousands of customers, essentially none of whom have any connection to the store, but whose identities are important to keep safe.
- Obviously, the web store needs some way to protect online transactions, be able to authenticate itself to the customers, and be able to handle dealing with anonymous users through a web-based interface.
- The security here can be provided through a combination of public-key and symmetric encryption, along with some type of authentication system (such as the RSA authentication).
- As it turns out, this type of security is so common in networked applications that a generic security protocol was developed that provides almost all of the security that our example web-store application would need: **SSL**.
- **The Secure Sockets Layer protocol is a generic security protocol that secures basic transactions over a network through a combination of public-key and symmetric encryption, message authentication through cryptographic hashing, and server authentication through a Public-Key Infrastructure (PKI)**.
- SSL is very generic, as it is designed to be blanket security for all network transactions, and the generic nature of the protocol means that it is big if fully implemented.
- The second example application is a payroll system for medium to large sized corporations.
- There are a number of people that require different levels of access to the system, such as human resources staff, the accounting department, as well as managers looking to hire new employees.
- Unlike the web store of our previous example, this application will likely be deployed on a corporate network that is shielded from the Internet, behind a firewall or other secure interface.

Embedded Security Design (Cont.)

- The payroll system is also not subject to the anonymous user problem.
- Everyone accessing the system should be a known user, so that the accesses to the system can be logged, for legal or bookkeeping reasons.
- The payroll system has a very different set of requirements for security from the online storefront.
- If we were to design these applications and try to apply a generic security mechanism to them after they were partially or fully implemented, we would run into a lot of trouble.
- The interaction between the security and the application does not fit very well, since the security is just "bolted on."
- No generic mechanism can even hope to capture the wide range of possible requirements for all applications.
- This is the primary reason why **we should incorporate security into our designs at project inception**, rather than try to add it in later.
- The requirements analysis phase of the design of a networked application should include all of the security features that will be needed in the final product, and into the foreseeable future.
- This seems like a very tall order, but when you compare the cost of maintaining an insecure system (which will be vulnerable to many attacks, adding to the cost of maintenance) to the cost of a more complete design up-front, good design wins hands-down.
- Obviously, with any design, **there are tradeoffs between cost, performance, time-to-market, and security**—all of these aspects of the application work against one another.
- **Higher security means higher cost and longer time-to-market.**
- **Higher performance often implies higher cost or lower security.**
- **Short time-to-market implies more cost, less performance, and less security.**

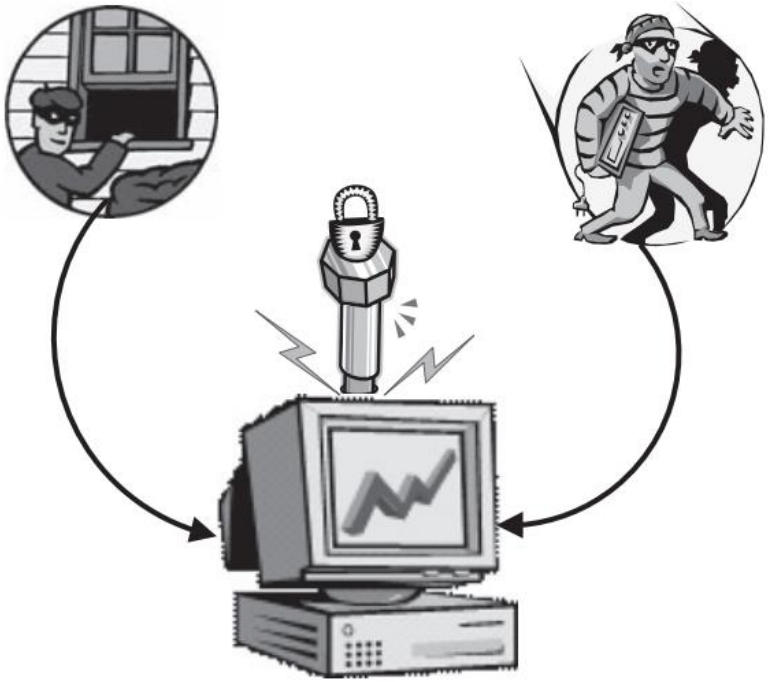
Embedded Security Design (Cont.)

- In our example applications, we have some very definite tradeoffs.
- In the first example, the web store, time-to-market and performance are likely important, where cost and security are not as important (this is reality, security suffers because of economics).
  - For this reason, the designer of our web store system will likely choose a generic security package to bolt into the application, which improves development time, but security may suffer from not taking the time to analyze the specific needs of the application (a web store will have the horsepower to use a generic solution).
  - Depending on the security package, cost may or may not be affected.
  - If the designer chooses an open-source package, the fee is likely minimal, possibly even zero.
  - Companies that do this type of development are likely to experience problems down the road because they did not take the time to understand the special needs of their specific application.
  - This is likely why we hear about companies "losing" thousands of credit card numbers or Social Security numbers to online bandits.
- In the second example, the payroll system, we have a different set of requirements.
  - In this case, there is likely a system in place already, and the company commissioning the new system is looking to upgrade.
  - In this case, cost and security are likely the most important considerations, with performance and time-to-market taking a back seat.
  - That's not to say that the application should be slow and take forever to implement, but the application needs to be secure, and it does not generate revenue directly for its owners, so the cost and security are just more important.
  - With this application, one or two dedicated engineers would likely concentrate on developing a thorough analysis of the security the system needs, taking into account access permission, the environment in which the application runs, the hardware on which the application will execute, and other factors, such as probable types of attacks and system failure.

Embedded Security Design (Cont.)

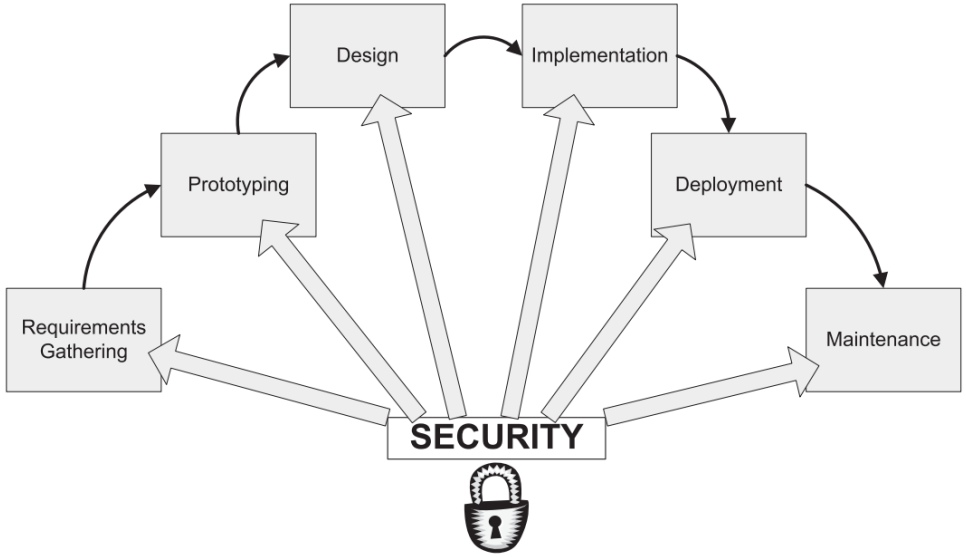
- It is also a system that will handle a large amount of money, so there are likely practical and legal issues that need to be considered as well, such as how to store the information safely and restore it after a system failure, and keeping the system up-to-date with any laws and regulations in the region where the application is deployed.
- Determining the security requirements for applications like those in our examples can be a daunting task.
- By definition, **a successful attack is usually something you did not think about when you built the system (if you knowingly put a vulnerability into an application it is called a backdoor)**.
- A good security expert should be critical of every component of a system, since every piece, every user, every function can be a weak point that can be leveraged to exploit that system.
- What can we do to make our lives easier so that we can develop secure systems without sacrificing all of our free time? → **The answer is simple (sort of): Keep the application simple**.

Embedded Security Design (Cont.)



Sometimes it's easy to avoid bolt-on security

Bolt-on Security Isn't Always Ideal



Steps to Designing in Security

Embedded Security Design (Cont.)

	Webstore	Payroll System
Users	Thousands, anonymous	Dozens, with usernames
Security	Protect credit info	Protect all personal info
Data lifespan	Weeks	Years
Biggest threat	Hackers on Internet	Disgruntled Employee
Primary Interface	Web Browser	Proprietary
Performance	High	Low to Medium
Primary Security Protocols	HTTPS/SSL	SSH, AES

Web Store vs. Payroll System Security

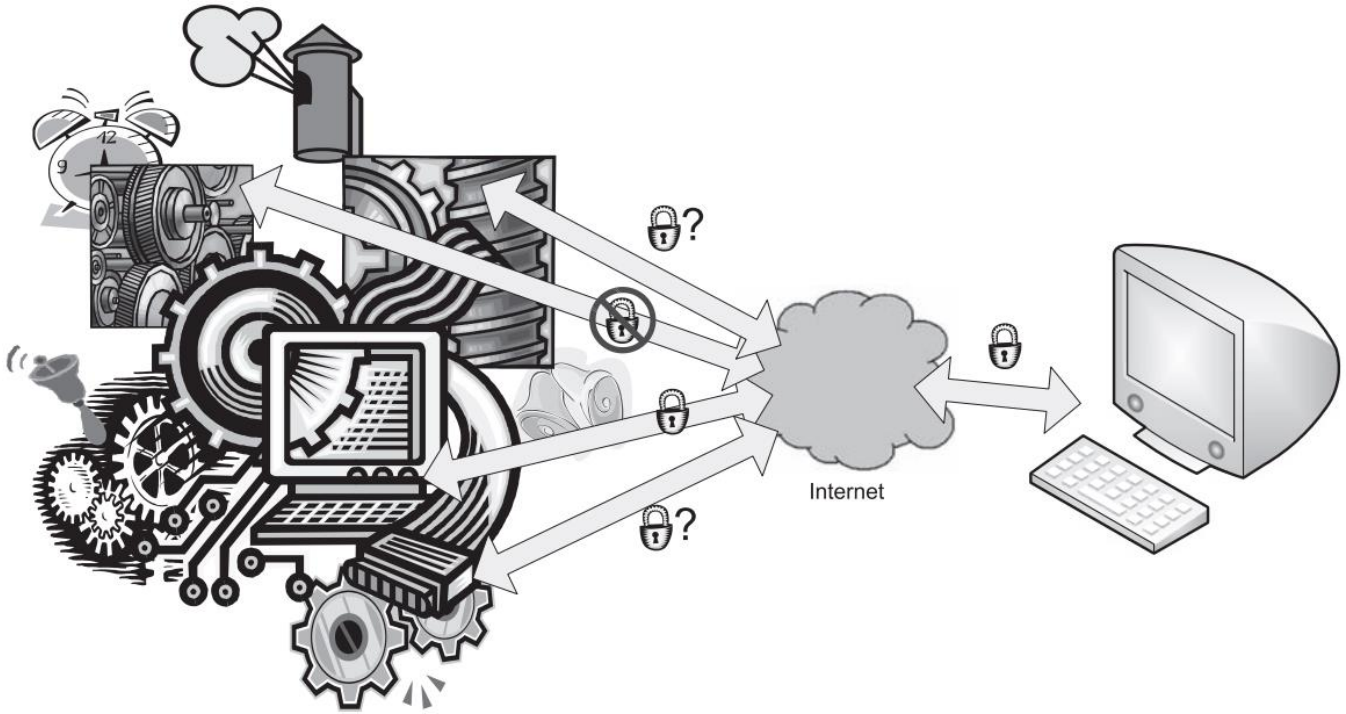
1) Support dozens of users with audit trail
2) Value security over quick time-to-market
3) High level of security
4) Low to medium performance is acceptable
5) Keep and protect data for years
6) Console-based or proprietary interface

Web Store Requirements

1) Support thousands of anonymous users
2) Quick time-to-market
3) Medium to high security
4) High performance and throughput
5) Short-lived data support
6) Web-based interface

Payroll System Requirements

Embedded Security Design (Cont.)



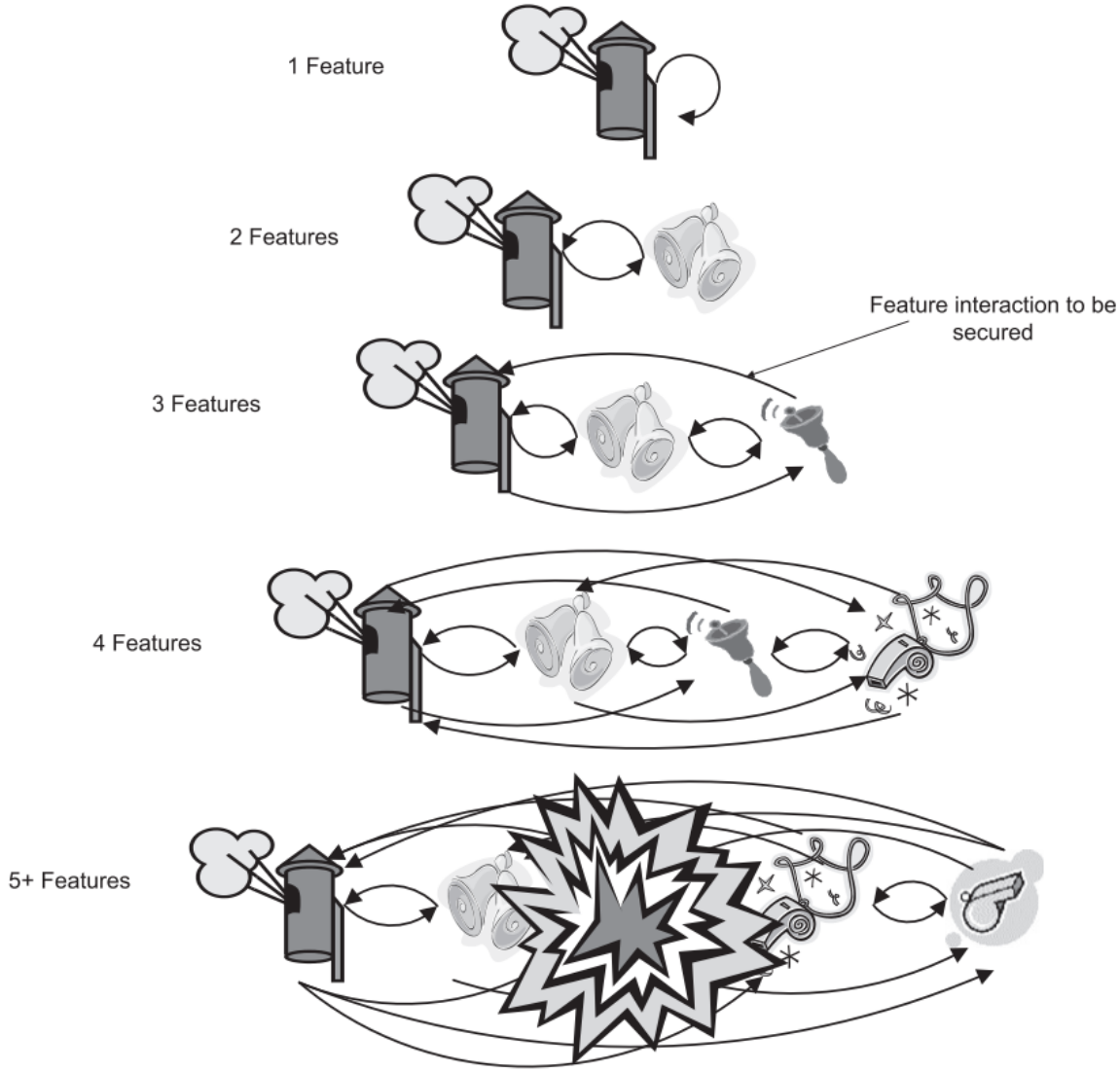
A complex system is much harder to secure than a simple one

Simple Makes Security Easier

The KISS Principle

- Many readers have likely heard of the **KISS Principle** - Traditional Definition: **Keep It Simple, Stupid** - referring to the fact that simple ideas are easier to grasp. → **Referring to electronics, it usually means that the simpler a system is, the easier it is to design, release, and maintain**.
- Keeping it simple also has another benefit, which is why we have **our own version of the KISS acronym**: **Keep It Simple and Secure**.
- An easy-to-understand system is easy to make robust, since all the problems are likely known ahead of time or are easy to predict.
- **Robust systems are easier to secure, since there are fewer variables to keep track of when protecting resources**.
- One of the greatest pitfalls in securing a system is the "**Feature**."
- For our purposes, we will throw out the marketing definition of and **define a feature as any singular task that the application performs (i.e., adding an entry to a database or reading a value from a field in a form)**.
- Unfortunately, the most secure system would have no features at all → It would be trivial to secure, since there is nothing it can do anyway!
- However, if we understand that **adding features increases the complexity of securing a system exponentially**, we are going to be much more careful about the features that we choose to support.
- **The reason for the exponential growth in features is due to their computing tasks as well as the interactions between features, both intended, and more importantly, unintended**.
- The problem comes in when we have so many features that the possible permutations of feature interactions grows so large it is not practical or feasible to model them all.
- The security ramifications of each additional feature can save time, money/cost, and headaches in the future of the project.
- Any features that can be removed at design time should be, and if possible, the system should be split into several different components/modules/computing features.

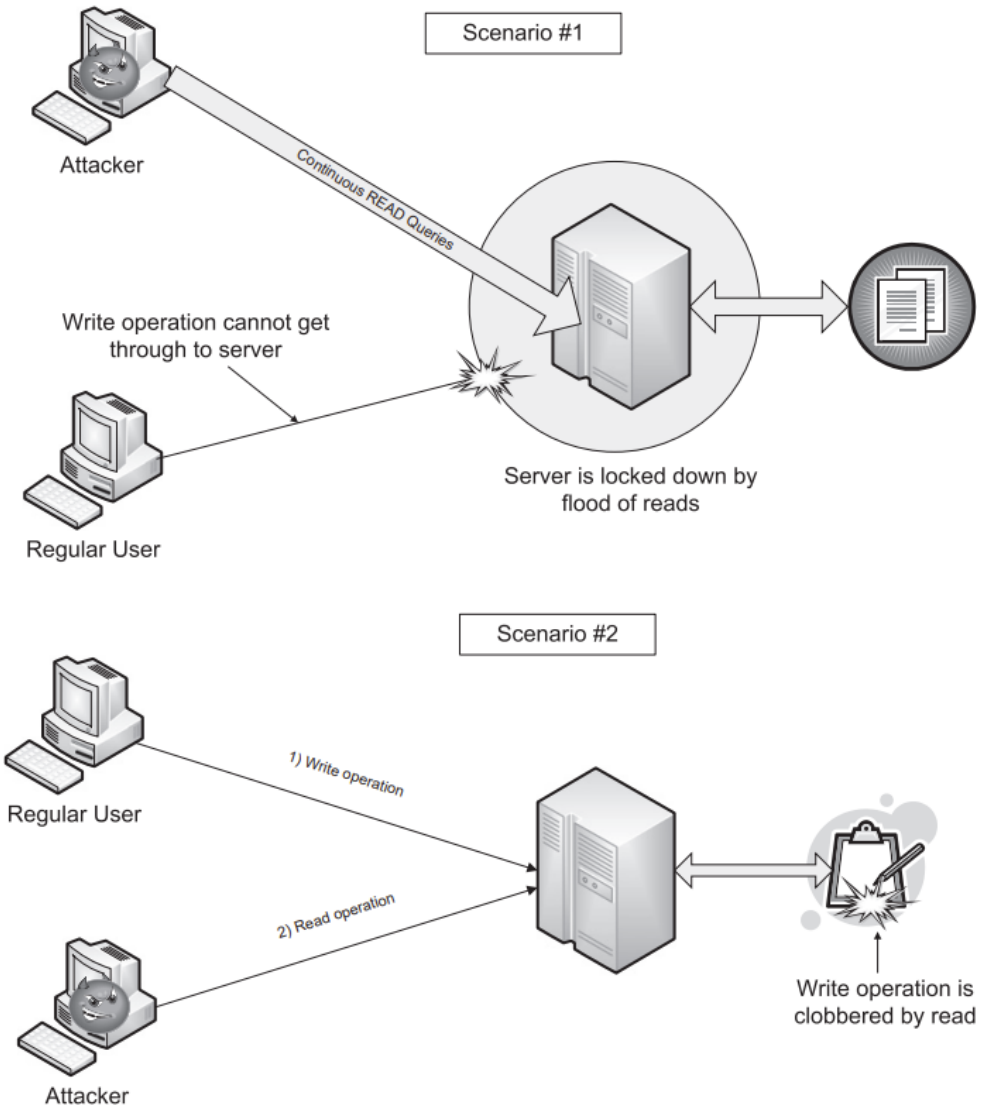
The KISS Principle (Cont.)



Feature Explosion

The KISS Principle (Cont.)

Inadvertent Read Behavior Prevents a Write from Happening



Modularity Is Key

- **Modularity** is a cornerstone of good programming practice, and it can be extremely beneficial in securing an application.
- If you **build small, robust, independent functional units**, and test each one extensively, then you have some guarantee about the **security** of each of those units.
- Trying to guarantee anything about a monolithic application is a lesson in futility.
- A good modular design also has the benefit of **flexibility**.
- Implement as many independent units as you can, **exporting a uniform interface**, and you can mix and match the features depending on the application.
- For embedded systems with limited resources, this is a definite advantage, since devices can have targeted responsibilities without having too many extra features and their resource requirements.
- Once you have **a number of robust, fully tested "feature units"** and you are building your application, you can focus on securing the interactions between those units, with some guarantee that they will behave well.
- To secure interactions between the modular units, it helps to have well-defined interfaces.
- There are several rules that you can use to keep your interfaces well-defined and less likely to be the cause of problems later.

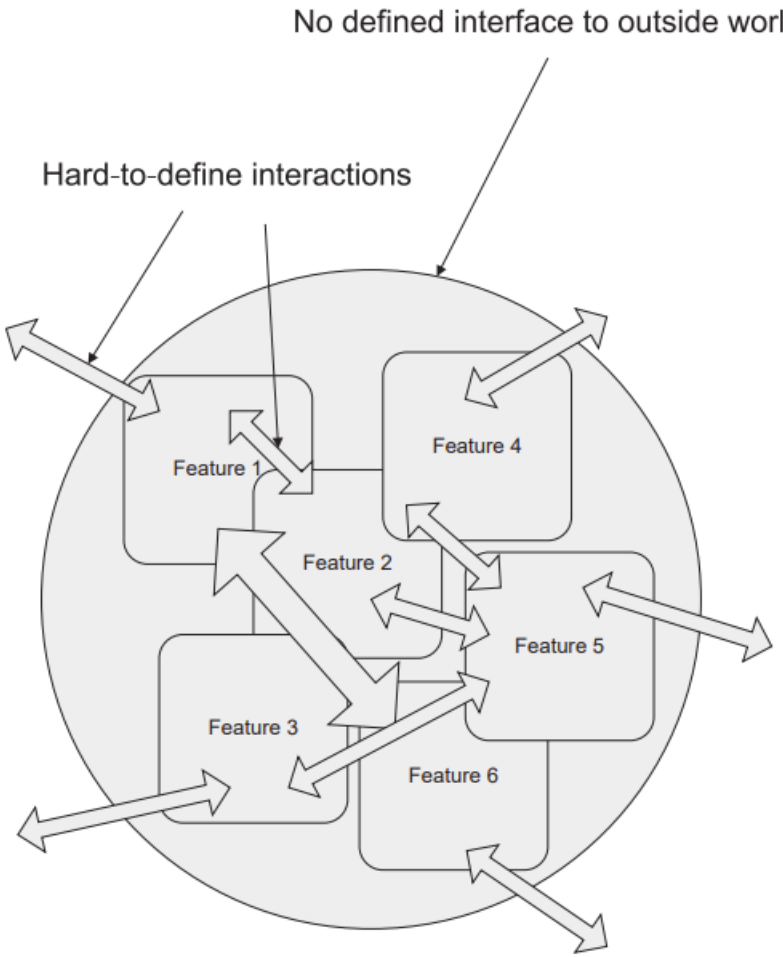
Modularity Is Key (Cont.)

- **To prevent errors from leading to a security breach**:
  1. If you are **passing a buffer of information** as **a parameter** to an interface function, always make sure that the buffer has **an associated length**, usually a separate parameter. → The functional unit should also check that length and generate an error if the buffer length is not within a known range.
  2. **Avoid passing pointers** (in C/C++), if possible. → Pointers by their very nature can be quite dangerous. → If the pointer is corrupted (either by accident or by an attacker), it can lead to compromise of the system. → If possible, limit the pointer to **a particular region of memory**, and abort an operation if that pointer ever refers to an address not in that region.
  3. **Avoid multiple levels of indirection**. → One level of indirection is hard to understand, two is even harder, and three is nearly impossible to comprehend in a complete sense. → Sometimes two or more levels are required to simplify certain algorithms, but **they should be carefully controlled** using a memory protection scheme (as mentioned in the previous item).
  4. **Range-check everything**. → This includes **flags, numeric parameters, strings, and pointers**. → Any variable that can misbehave is a potential vulnerability, so define each variable's expected values at design time, and be sure to check that each variable never deviates from its expected set of values.
  5. **Assert** is your friend. → **Using "Assertion" Function** → If you don't have the assert function, create your own that prints out information when something goes awry. → Identify anything in your application that has **a hard/critical boundary**, and protect that boundary with an assert. → It may seem a bit cumbersome, but any **assert failures** you catch and fix during development and testing will prevent potential security breaches after deployment.

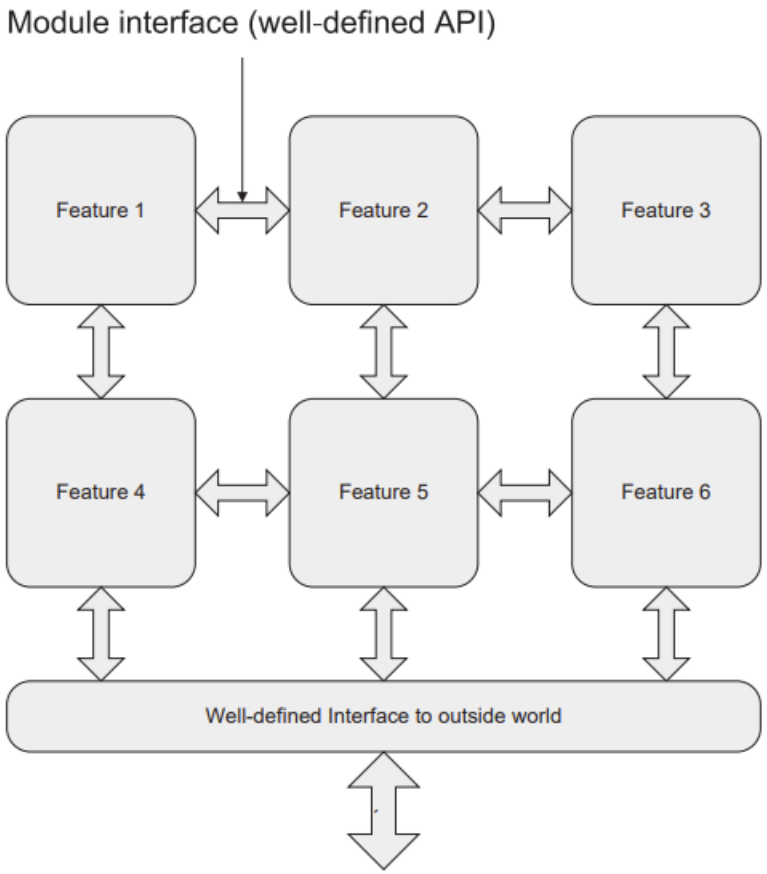
Modularity Is Key (Cont.)

- **Overheads**: **range checking** adds additional logic, **limited use of pointers** can cause extra code bloat, and **passing a length along with a buffer** is frequently cumbersome and uses up both code and data space.
- **So how do we reconcile these needs with our constrained system?** → The most obvious solution to the aforementioned issue is to remove all range checking, use pointers everywhere, and just assume that callers passing buffers all obey the rules. → This is unfortunately the easiest solution to the problem, and therefore is likely one of the most commonly used, even though it can violate some basic rules of software engineering.
- An application that strips all of these features can still be fully functional, and is likely much smaller than its **more reliable counterpart**.
- Given this, we need to think about security very early on in **the design phase of an application** so that we can account for the additional resource requirements needed to implement that security.
- This is a good thought, but how do we really guarantee that a design will lead to a secure, reliable application that does not hog resources? → The best solution to making an application smaller is to remove features.
- Most marketing types may balk at this, because the addition of extra features often sells more products, but there should be at least some level of compromise.
- Remember that each additional feature can add an exponential amount of complexity to the system, making it that much harder to have any guarantee that the system is secure.
- The **most secure** system has **zero features**, since there is nothing to exploit, but that system also has no practical purpose, since it cannot do anything useful either.
- We should aim to develop applications with fewer features if they are to fit into a small, dedicated embedded system.
- Think about which features are absolutely necessary, and which are **expendable**.
- Try to define the features as well as possible, and nail down the requirements to the minimal set possible that still meets the goals of the project.

Modularity Is Key (Cont.)



Interactions in a monolithic application are difficult to determine



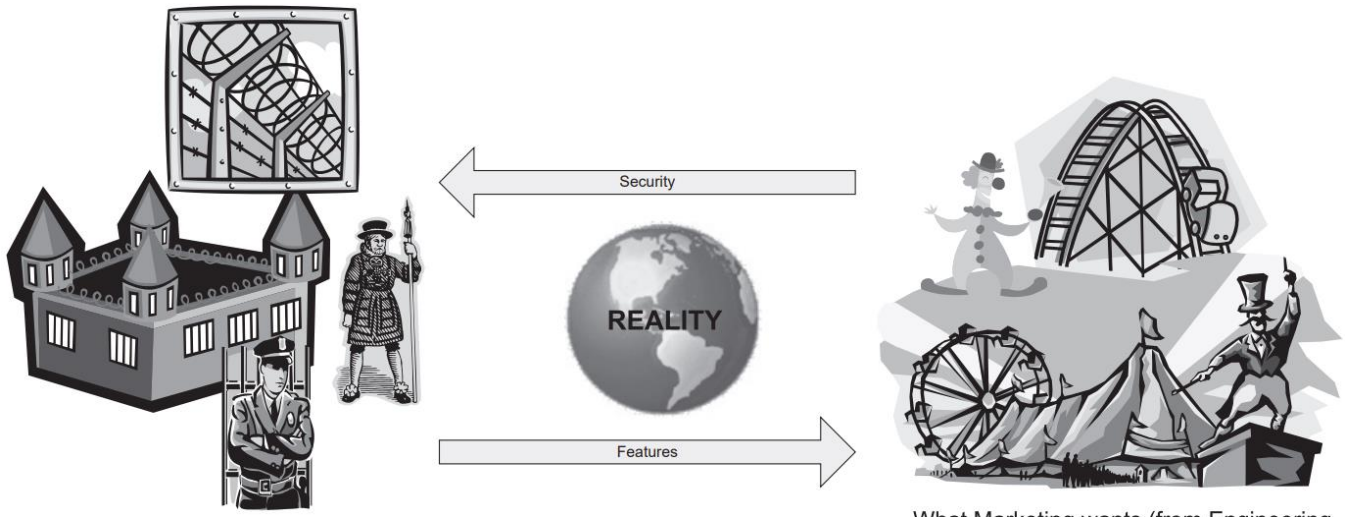
A modular application with a well-defined interface is easier to understand

Modularity Is Key (Cont.)

API Rules

1. *Always* make sure buffers have an associated length when passing as a parameter. In the function, don't access the buffer past that length.
2. Avoid passing pointers (in C), if possible.
3. Avoid multiple levels of indirection.
4. Range-check *everything*.
5. Identify invariants and enforce them with assert.

Marketing vs. Engineering



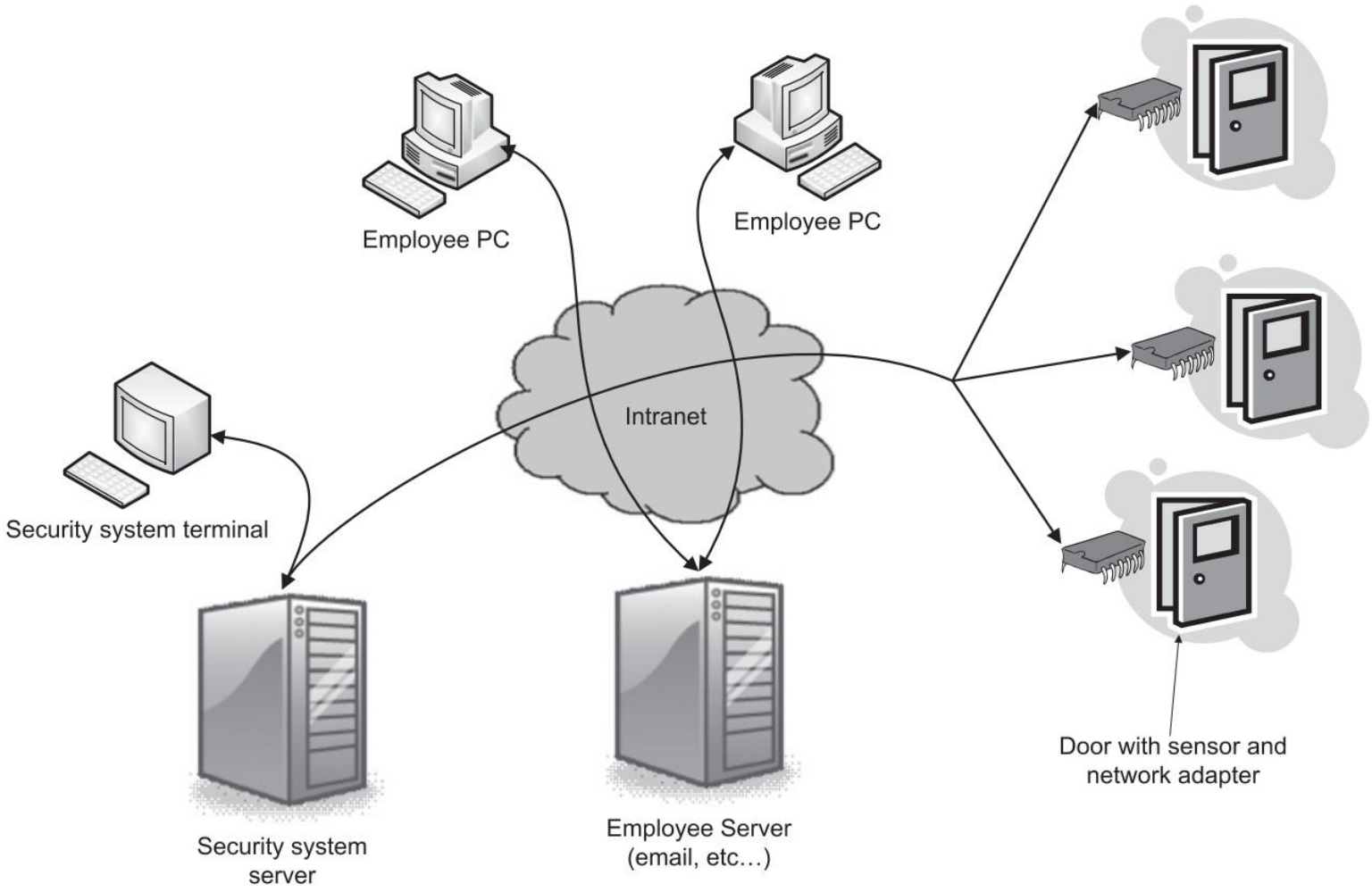
What Engineering wants (from Marketing perspective)

What Marketing wants (from Engineering perspective)

Modularity Is Key (Cont.) – A Sample Application

- A door monitoring system connected to an internal network.
- Every door has **a small monitoring device** consisting of a microcontroller connected to a keypad and a magnetic sensor that determines whether the door is open or not.
- Each microcontroller is equipped with an Ethernet interface that is connected to an intranet within the building, and a centralized server provides data logging, control over the door locks, and other services.
- In order to cut down on cost, the intranet is shared by the company's employees so that both they and the door monitoring system utilize the same network resources.
- It is pretty clear what the embedded devices should be doing— checking whether the door is open or not, verifying key codes with the central server, and controlling the door locks.
- Any additional functionality should be considered potentially detrimental to the security of the system, because those features would be strictly unnecessary for the proper functioning of the application.
- Removing these extra features frees up additional resources for the application at each node, allowing for more boundary checking, free space to remove indirection, and has the benefit of reducing potential security holes.

Modularity Is Key (Cont.)



Door Monitoring System

Modularity Is Key (Cont.)

Sample Door-Monitoring Main Program

```
main() {
 byte key_flag, alarm_flag, open_flag;
 byte key_value, key count;
 word open_count;
 Key_flag = alarm_flag = open_flag = 0;
 Key_value = key_count = 0;
 Open_count = 0;
 Init_Network();
 Init_Door_Sensor();
 while(1) {
 Key_value = get_key();
 if(key_value != -1) { // -1 indicates no key pressed.
 Key_array[key_count++] = key_value;
 if(key_count >= 5) {
 Have_key = 1;
 }
 }
 if(have_key) {
 Open_flag = check_key(key_array);
 Have_key = 0;
 }
 if(open_flag) {
 Open_Door();
 Open_flag = 0;
 }
 // Do other stuff (monitor door open/closed)
 }
}
```

Modularity Is Key (Cont.)

```
// ISR – key_value contains the value of the key following the interrupt firing
void key_isr() {
 Key_buf[cur_key_pos++] = key_value;
}
// get_key function – returns a word containing the value of the next key press
//following the previous call to get_key
word get_key() {
 word ret_val = -1; // Default = no key available
 If(cur_key_pos > sizeof(key_buf)) {
 // circular buffer – reset current position in buffer
 Cur_key_pos = 0;
 }
 if{last_key_pos != cur_key_pos) {
 ++last_key_pos;
 If(last_key_pos > sizeof(key_buf)) {
 // circular buffer
 Last_key_pos = 0;
 }
 ret_val = key_buf[last_key_pos];
 }
 return ret_val;
}
```

Sample Code for Keypad Interface

Modularity Is Key (Cont.)

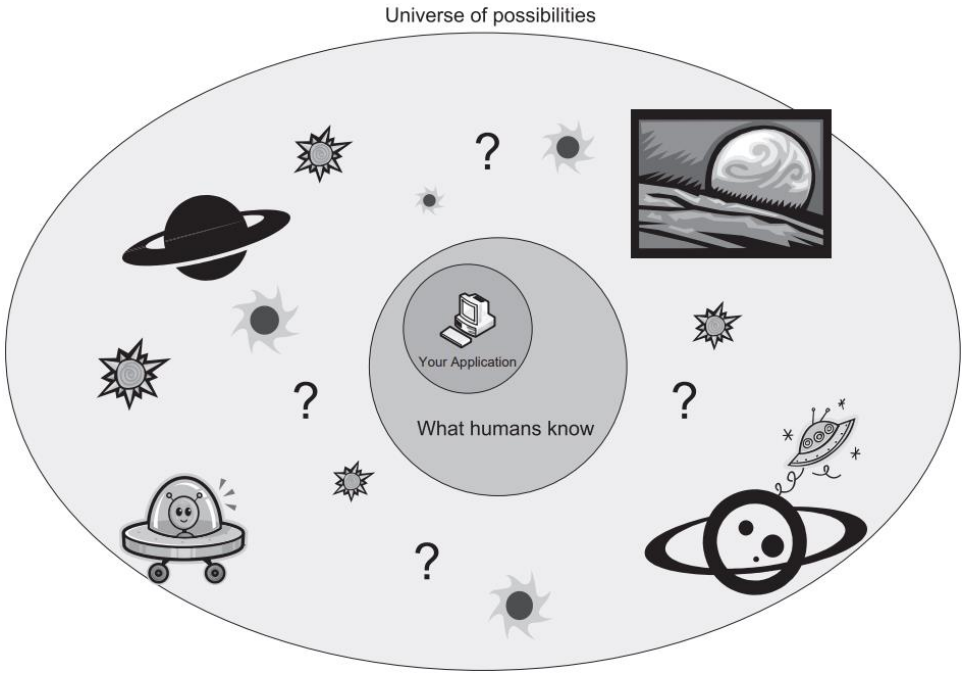
```
// Static data
char buffer[100];
char open_flag = 0;
// Read from a particular SSL socket into a static buffer
void read_message(SSL_sock_t ssl_socket) {
 // Reads message from the SSL socket into the static buffer if
able
 if(ssl_ready(ssl_socket)) {
 // Writes into buffer current SSL record contents
 ssl_read(ssl_socket, buffer);
 }
 … // Do other stuff
}
// Reads from an SSL socket into a buffer (passed via pointer)
void ssl_read(SSL_sock_t ssl_socket, char * buf) {
 int bytes_ready;
 bytes_ready = ssl_bytesready(ssl_socket);
 ssl_sock_read(ssl_socket, buf, bytes_ready);
}
```

Static Data and Receive Functions

Modularity Is Key (Cont.) – Well-Defined Interfaces

- One of the key features of a robust modular system is **the existence of well-defined and restricted interfaces between modular units**.
- A **well-defined interface** includes well-understood pre and post conditions, definite limits on variable-length parameters, no "extra" parameters, and it has good documentation so that it can be understood by the developer working with it.
- The last point here is often overlooked, since there is often an implicit assumption that anyone using the API can figure it out from a terse function reference.
- There is also an implicit assumption that the API functions are well behaved and generally can be trusted.
- Trust API functions, only as much as you have to. → It never hurts to wrap a call to a function that is not yours with some extra checking to be safe (using assert operation).
- **Check API Function**:
  - **Check Pointers** → A well-defined API function should utilize type checking (if available) to protect the pointer in the function, such as utilizing the const pointer modifier available in C++ language.
  - **Check Buffer Length** → Utilize functions that require a length parameter if a buffer is to be passed in as a parameter. → The presence of the length parameter indicates that the function is not relying on some inherent property of the buffer to know when to stop writing to it.
  - **Analysis of Code and Data** → The more you understand the code you write and all of its computations (intended or not), the more secure your application will inherently be.

Modularity Is Key (Cont.)



There Is a Universe of Possibilities Beyond What We Know

What You Cannot See Can Hurt You

Pick and Pull

- Things protocol designers never intended → Things you can do to your program that would void your warrantee if one existed. → **Solution**: Follow certain rules for not causing any problem.
- **RULE #1: Removing a feature can be safe, but make sure you understand it first.**
  - One of the first things you realize when you dig into most protocol implementations is that you are almost always dealing with a state machine of some type.
  - If you can start to understand how the protocol works and what the states are in the state machine (it helps to look at the actual protocol specification), then you can start to see what parts are not strictly necessary.
  - One easy way to spot "**features**" that you can remove is to look through the specification for the words "**optional**" and "**should**"
  - You can also look at some of those things in the specification that are labeled with "**must**" and see if it really "**must**" have that feature.
  - Often, the "**required**" features are needed to meet a protocol specification, but are not strictly necessary for the protocol to operate.

Pick and Pull (Cont.)

- **RULE #2: Don't ever try to outsmart cryptographers. They are smarter than you.**
  - As a general rule, it is usually a good idea to always assume cryptographers are smarter than you (unless you are a professional cryptographer).
  - Sometimes there is a license or royalty attached, but for the most part, you can get your hands on at least a pseudo-code implementation of any number of powerful encryption algorithms.
  - There is no shame in using existing algorithms and protocols to secure your application, but **do not**, under any circumstances, try to optimize or **remove parts of an encryption scheme** (unless you are, in fact, a cryptographer).
  - The algorithms that you hear about, 3DES, AES, RSA, these are all proven, tested algorithms that rely on certain characteristics of the math on which they are based.
  - "**Security by obscurity**" is a tempting method to keep your data protected, but as has been proven with many systems throughout history, it is not too difficult to discover how a cipher works without knowing its actual implementation.
  - Existing algorithms are compatible with other systems, and it makes communications easier to implement, since you can use off-the-shelf applications and mechanisms with your application.

Pick and Pull (Cont.)

- **RULE #3: You do not need everything that every protocol offers.**
  - In the world of the PC, you have nearly infinite resources with which to play.
  - Since the PC is the current dominant entry point to the Internet, it makes sense that the myriad of protocols in existence today are designed around the PC's basic specifications: **loads of memory, a giant hard disk, and clock speeds measured in gigahertz**.
  - What the embedded designer should be looking for are the parts of protocols that are specifically useful for a particular application in **embedded network**.
  - An embedded application is dedicated to a specific purpose and does not need all the extra functionality - it only has to be compatible with the other parts of the application.

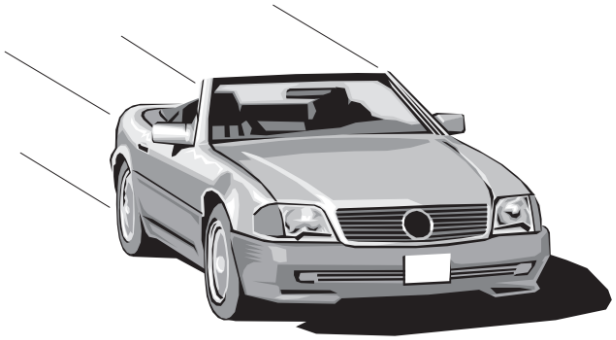
- **RULE #4: Make sure you only apply "safe" optimizations to security protocol implementations.**
  - When you are implementing a security protocol for **a resource-constrained system**, it will often be tempting to try and optimize parts of the protocol that appear to be slowing things down or using up too much memory.
  - **Be extremely careful in what you optimize and how you do it.**
  - Every tiny detail of the algorithm has been designed to provide the security guaranteed by using that particular algorithm.
  - All the constant values seen in any cryptographic algorithm implementation are carefully selected for
  - What we need is a concept of **"safe" optimizations**, transformations that can be applied to the code without breaking any of the assumptions inherent in the algorithm.

Pick and Pull (Cont.)

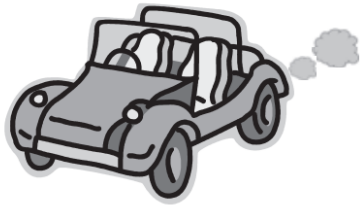
- **RULE #5: Simple is better.**
  - **Simplicity** makes an application **easier to understand**, and therefore **easier to make secure**.
  - Don't overcomplicate your application with lots of extra protocols and features.
  - List the requirements of the application and cut out any that are not absolutely necessary.
  - Being an embedded systems designer, you should already be doing this, so this is a very complementary operation.
  - Keep the feature set down, and you will have a smaller application.
  - Having a smaller application means you can study it more and make it more robust and secure.
  - If new features are being added or old features are being expanded as development continues, you are introducing a delta to your existing code base that makes it even more difficult to analyze what you have done.
  - On top of that, the extra time taken up by implementing the new features could have been used for tightening up what you had already implemented.

Pick and Pull (Cont.)

Resource-Constrained Applications Are Utilitarian

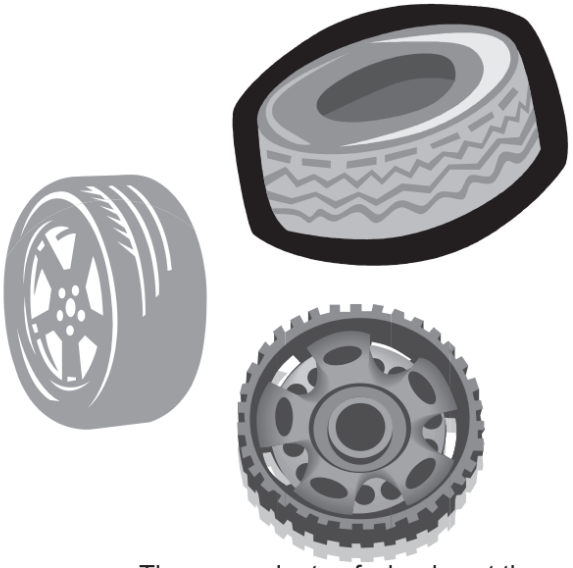


Horsepower is good for some applications



Resource-constrained applications are not flashy

Don’t reinvent the wheel...



There are plenty of wheels out there to choose from!

Don’t Reinvent the Wheel

Justification

- We want to be able to build devices that have all the properties of their more expensive brethren for a fraction of the cost to open up new opportunities and new markets.
- In order to achieve this, we have to be a little creative about how we go about implementing our applications.
- **We can't be wasteful like PC developers** → You may remember that a megabyte of RAM was plenty enough for an old DOS machine).
- Instead, we have to conserve every shred of resources.

Assignment

- **Reading Assignment:**
  - Stapko, T., 2011. **Practical embedded security: building secure resource-constrained systems**. Elsevier.
    - ✓"Chapter 5: Embedded Security", Pages 83-114.

GANNON UNIVERSITY

Questions?