IoT Security
Top 20 Design Principles

Empowering Trust™

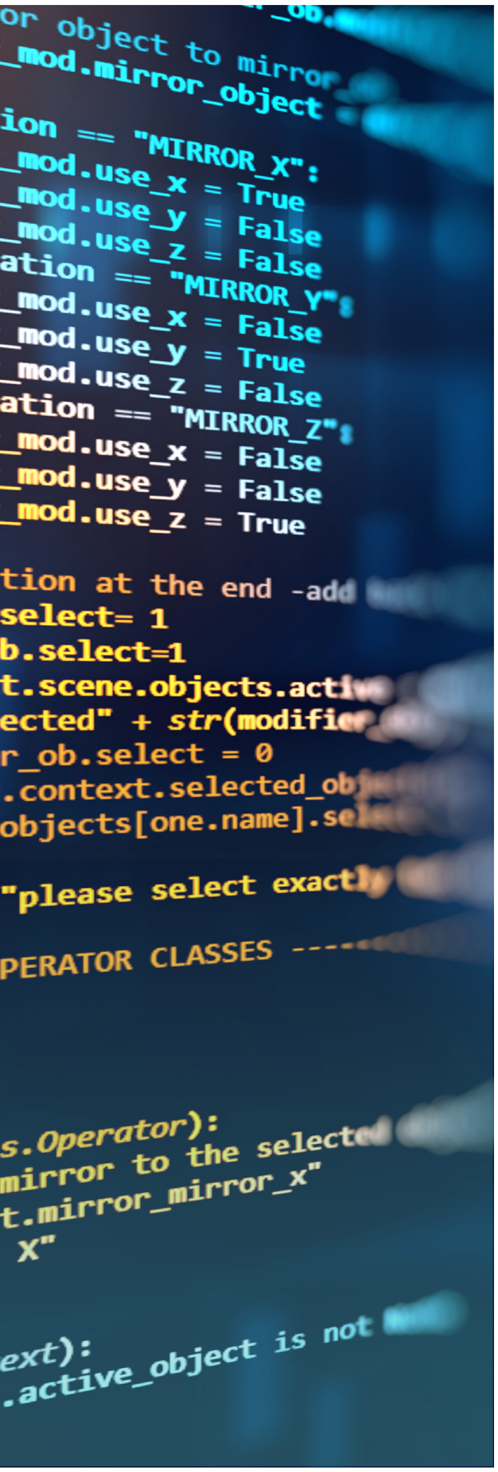
Executive Summary

In a competitive market, it is important to find a differentiator for your products, and often a competitive advantage is sought through adding smart features and connecting products to business networks or the internet. However, as new features and connections are added, the security of such systems is often degraded.

Additionally, the security of connected products is increasingly becoming a matter of organizational and even national interest. Malware that can take control and subvert the operations of connected systems has been used to launch some of the largest attacks ever seen on the internet. The connected nature of these systems also means security must be considered for any apps that run on separate systems, such as cloud services and consumer phones.

Of course, fitting security into increasingly tight time and budget requirements for product development can be difficult. Fortunately, there are some simple steps that can be taken to increase the security of connected systems. These outlined steps are organized with the most important requirements first, and it is recommended that these are addressed as the initial priority for all aspects within a system – product, system, cloud and app.

Top five priorities


```
or object to mirror
_mod.mirror_object
ion == "MIRROR_X":
_mod.use_x = True
_mod.use_y = False
_mod.use_z = False
ation == "MIRROR_Y":
_mod.use_x = False
_mod.use_y = True
_mod.use_z = False
ation == "MIRROR_Z":
_mod.use_x = False
_mod.use_y = False
_mod.use_z = True

tion at the end -add
select= 1
b.select=1
t.scene.objects.activ
ected" + str(modifier
r_ob.select = 0
.context.selected_ob
objects[one.name].se

"please select exactly

PERATOR CLASSES ----

s.Operator):
mirror to the selected
t.mirror_mirror_x"
 x"

ext):
.active_object is not
```

1 Provide a manual override for any safety-critical operations

Advanced features are great when they work. However, sometimes these features fail – often through no fault of the system itself. The local network of the customer can be poorly configured, or their internet connection may become interrupted. In such cases it is important that any lack of functionality this causes does not result in a safety problem for the end user. Examples may be providing a physical key back-up for a smart door lock or a manual override and safety-limiting feature on an IoT thermostat.

2 Ensure parameters which could compromise the system (secret or private cryptographic keys, passwords, etc.) are unique per device

Passwords that are used for security-sensitive features should be unique per device. A password that everyone knows is not much of a password. There are simple solutions for this, however. For example, secure passwords can be randomly generated and printed on the device's serial number sticker. If the device is not going to be easily accessible during normal operation, consider providing such a sticker inside the manual or quick start guide, which can be taken out and put somewhere the user will not forget. Of course, there will always be scenarios where customers forget or lose passwords, so system recovery methods, such as a physical reset button which enters a password recovery mode when pressed, should also be securely implemented.

Any secret (symmetric) or private (asymmetric) cryptographic keys should also be managed as unique for each device or application. There are a few ways in which cryptographic keys can be determined even when they are apparently being stored and managed inside a secure device. Often these methods are not worthwhile for extracting keys from a single device, but if the same key is used in thousands of devices, the economics of such an attack change considerably.

3 Test the system to be sure it is free of known, exploitable vulnerabilities prior to release

The software in connected devices, applications and cloud services are often comprised of various software components, including existing software (such as open-source code and third-party libraries) as well as commonly used protocols and functions (such as databases). Each of these software components may have its own vulnerabilities, and it is important that a check is made for known vulnerabilities before releasing any system.

This can be achieved through various software utilities and scanning services for cloud-based systems. In the Payment Card Industry (PCI), for example, look for vendors who have passed the requirements from ASV, which has done a good job of providing a minimum validation of scanning vendors.

4 Allow for software updates and ensure they are cryptographically authenticated prior to installation and execution. Implement anti-rollback features to prevent the installation of previous vulnerable versions of firmware

No matter how well software is designed or tested, there will always be bugs and vulnerabilities that are missed or discovered after the product ships. It is important to allow for the update of the software to ensure that it can be patched when any such bugs are found. However, if this is not carefully implemented it can lead to additional vulnerabilities where a bad actor can install their own software into the device to prevent its normal operation.

To prevent this, software updates should be cryptographically authenticated. This is often achieved through the use of a digital signature on the system firmware image, which can be checked by the original firmware (or bootloader of the device) prior to installation. Using a digital signature based on a public key algorithm (such as RSA or DSA) ensures the devices themselves do not need the part of the key (private or secret) that is used to generate the authentication data.

If a symmetric key system is used instead, such as a (H)MAC, this secret key is needed in each device. This means that access to the firmware in one device provides the ability to create valid firmware signatures for any device (unless there is a unique key per device, which is not feasible for IoT systems). So, public key cryptography is strongly recommended.

It is also important to include methods to prevent a bad actor from installing a previous version of firmware, which would reinstate any otherwise patched vulnerabilities. This can be done by including an increasing number (monotonic) in each firmware release that is checked before install to ensure that the firmware version attempting to be installed is not older than the current version in the device.

5 Use industry standard security protocols with best practice defaults for any remote or wireless connections and authentication of connections to management services

It is vital to protect communications which may be subject to interception or modification using an industry standard security protocol such as TLS or WPA2. Additionally, the exact use of the security protocol is also important – for example, it is possible to use TLS and still be insecure if it is not configured correctly. These protocols allow for the authentication of connections, but only when implemented properly. Therefore, validation of certificates or certificate pinning should be used to ensure that the connection is both secure and private.

Use the latest version of the protocol and software library, and monitor any changes so patches can be provided when problems are fixed. The use of the security protocol should cover all communications where possible, regardless if they are security-related or not. This will make compromise of the system more difficult, as any bad actor must first compromise the security protocol before gaining access to try to compromise the device.

This requirement also covers the use of wireless protocols, where the use of security protocols such as WPA2 is equally important. It may be necessary to allow for customers to disable security features, but consider providing them guidance on why this is not recommended.

Best of the rest checklist

6 Do not store passwords in clear text

It is common knowledge that people tend to reuse passwords, so a password extracted from a single compromised system may be useable on other systems, accounts and online services. Instead of clear text, passwords should be stored using a “one-way,” and computationally intensive, algorithm such as BCrypt.

For any cloud environments, this item should be considered part of the top five requirements.

7 Authenticate remote access and system management interfaces with session and time-out limits

Access into a system, whether to a device from outside the local network or into a cloud system, should be authenticated to prevent access by unauthorized parties. For back-end or cloud-based systems, consider implementing two-factor security measures such as those provided by FIDO-compliant tokens and software. SMS-based One-Time Passwords may be implemented, but these are under increasing attack and newer, more secure methods are recommended.

For connection into local networks from an external system, consider VPN connections or routing data through a TLS connection tunnel that can provide authentication (and where certificate validation/pinning is performed as required). Local connections may require only a password, but may also provide authentication through physical proximity, such as a Bluetooth/NFC connection or through a physical button that must be pressed to access the service. If localized wireless is used, ensure security best practices are followed.

Debugging interfaces, such as JTAG and in-circuit emulation, should always be disabled on production devices. Although accessing such interfaces requires local physical access, enabling them greatly simplifies the work of a bad actor in developing attacks.

These management services may have a range of functions, from turning a camera to point in a different direction to loading new certificates, firmware or other security-related features. If a device has multiple features that can be accessed through remote administration features, consider providing different levels of authentication so it is possible to isolate security-related features from user features.

In addition, provide an absolute limit on the time during which any administration features can be accessed during one session. This prevents people from forgetting that they have left these features on.

8 Ensure cryptographic key methodologies generate sufficient randomness

Generating good random numbers is actually very hard, and the use of poor random numbers has been the source of many vulnerabilities. The root cause usually is based around two issues: first, computing systems are deterministic, meaning the same program, given the same inputs, will produce the same output every time; second, we as human beings are not very good at spotting a lack of randomness.

Input from an embedded device cannot be solely relied on to produce good random numbers. It is often best to take input from various sources. One of these may be a standard random function, but other sources also include the least significant bits of an A/D input, network traffic timing, hard-disk seek timing, millisecond data from a real-time clock source, user input timing, etc. These can then be combined and provided as a seed to a pseudo-random number generator, such as those outlined in NIST SP 800-90A.

Cryptographic keys should not be directly generated from passwords. It is better to use the password to enable access to the use of a key generated from a strong random number process.

9 Detail all customer data – including audio, video and personal details – that can be exported to cloud systems or third parties. Provide an opt-in for such collection

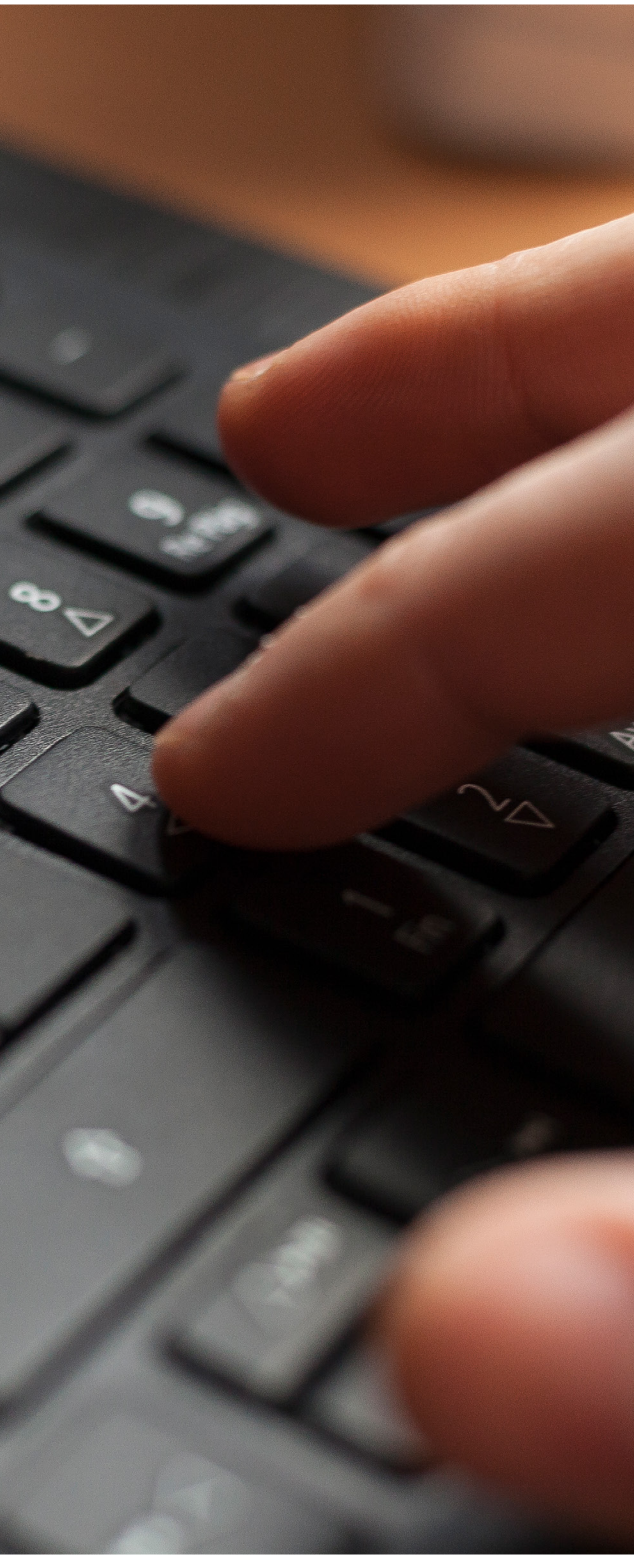
Many systems provide advanced features for processing in a cloud environment. However, not all consumers are aware of the collection and export of this data, and privacy concerns may exceed the customer desire for provided features. It is important that consumers are given control of the data they provide outside their own networks through clear disclosure and an opt-in rather than a default-on process.

10 Only use industry standard cryptographic algorithms and modes of operation for any security protocol (such as firmware authenticity checking)

Cryptographic algorithms are complex, and these days it is reasonable to say that there is no one person who is able to say that any particular algorithm is secure. The only way to have any confidence in an algorithm is to subject it to study from a host of experts over many years. Even then, new research and findings may come along and reveal flaws previously unfound.

Therefore, it is strongly recommended that only cryptographic algorithms, key lengths and modes of operation that are research-verified, and generally accepted to be secure, are used. A great reference for this is NIST SP 800 57, which basically defines Triple DES, AES, RSA and Elliptic Curve Cryptography (ECC) as the only algorithms for use.

The mode of operation, or the way in which the cryptographic is actually used to encrypt the data or provide authentication, is also important. It can be easily overlooked that using a simple mode of operation such as Electronic Code Book (ECB) can actually expose patterns in the plaintext data and may not achieve the security that is intended when implementing the encryption.



11 Provide ability for users to enable on-demand features they may not want or only use intermittently

All software has bugs, and many of these bugs expose potential security vulnerabilities. The more software a product has, the more bugs it is likely to have. Of course, these bugs can be minimized through testing, patching and other methods, but many bugs will remain unfound and unpatched until an exploit is released. In contrast to this, many products rely on an extensive feature-set to differentiate themselves in a commercially aggressive market. To balance these conflicting requirements, it is ideal to have some of the more advanced features disabled by default so a system can remain secure even if an exploit is found.

For example, remote access, wireless pairing and advanced functions (email, printer interfaces, etc.) may be provided but disabled by default and provided with a timed access feature so the customer can enable the feature only for a certain period.

12 Implement a power-on self-test that validates core functions and integrity of firmware prior to execution. Implement a cryptographic chain of trust from the hardware during boot where possible

Ideally firmware should be validated on each boot to ensure it has not been altered since being installed. This can be achieved when there is a signature across all firmware anyway, which may be the case if the system runs a simple function executive, but is significantly more difficult when it is a complex operating system (OS), such as Linux. Validating all of the different files, scripts, etc. that go into ensuring a complex OS runs correctly is complicated. Potential solutions include the validation of the device bootloader (from a hardware root of trust, which requires support in the processor being used) and then using that bootloader to validate an OS image which is unpacked and installed.

Of course, this all takes time. But it is useful to prevent things like Ransomware which may look to install software that renders a device inoperative until a ransom amount is paid.

13 Ensure that any system defaults, such as passwords, certificates or keys, are forced to be changed prior to initial operation

System defaults should be avoided where possible, but such defaults are often necessary. For example, a default may be required to allow for the boot-strapping of the system for the first time. This can be acceptable, but this default value should be forced to change as part of the overall setup.

Ultimately, defaults should only be considered for certificates and other such items which may be required to be present for normal operation, but must be changed by the user before installation and operation. This also covers any test values that may be in the firmware, and these should never be left in a production system.

14 Ensure error messages or responses to invalid messages do not expose sensitive data

When a system is incorrectly accessed, it is common to return some form of error message to indicate what error has occurred. Such messages can easily reveal sensitive information about a system and must be very carefully implemented. Consider returning only good/bad indications from production systems, at least until some form of debug state is activated (through an authenticated access, of course). Never return details of any decrypted data if there is a failure of some sort, even to note what error has occurred in the decryption and validation. Simply reject the connection in such cases.

15 Ensure cryptographic keys are only used for a single intended purpose

Key management (the way cryptographic keys are used) is extremely important. The cryptographic algorithm is only a part of the security. The way the algorithm is used, and the way the required cryptographic keys are used, is also vital.

To prevent compromise, it is good practice to use cryptographic keys for only one purpose. Data encryption keys should be used only for encrypted data. Keys used to secure passwords, for example, should be different. Do not mix keys (or key pairs) between uses for encryption and authentication. Each key should have a unique use.

16 Implement least privilege in all systems

Modern processors often offer different privilege levels, which include potential access to memory and other resources. These features can be used by the software to help secure assets within the device by ensuring only software with the right privilege can access them. The interface to the hardware-level privilege controls of a processor are often managed by the OS, such as Linux, but they can also be controlled even if a device does not use a complex OS (for example, if it uses a simple function executive, or cut-down RToS, instead).

Whenever possible, keep the use of root or supervisor-level privileges in embedded systems to an absolute minimum, and maintain secret data, such as cryptographic keys, at the highest level of privilege (hardest to access). Similar rules hold true for apps and cloud-based systems: keep the use of elevated privileges to a minimum, and try to isolate functions as their own user or privilege set.

17 Implement protections to prevent execution of data memory

Many remote attacks aim to gain execution of code supplied by the bad actor, which is provided through a specific vulnerability. It is almost impossible to avoid all vulnerabilities in code, but mitigating the potential for remote code execution through coding vulnerabilities is possible through a number of different methods.

Many processors provide “no-execute” functions, which can mark specific areas of memory that cannot be used to reference directly-executable code. Alternatively, some processors may even provide entirely different code and data memory segments, which makes direct code injection attacks impossible (although other types of attacks may still be performed).

In addition to direct hardware protections, there are also other protections that can be applied at the software level. Some of these can be provided by the OS itself and some may be applied through compiler settings when creating the object code to load into the device.

Determine what protections are possible on the various components of any system and implement as many as technically and operationally feasible.

18 Do not allow direct execution of externally-provided commands, scripts or other parameters that are not within the defined functions of devices

In addition to direct code execution, there are additional ways a bad actor may gain access to a system if there are other interpreters or execution environments available. For example, a system may allow for non-native code to be executed, such as JavaScript, which can then lead to potential vulnerabilities. While this does not mean it is always necessary to disable things like JavaScript completely, it is worth understanding the system's need for this type of functionality. If it is included, it will often require more complex security solutions to maintain the overall security posture of the system.

19 Create and compile firmware for devices so that it contains only code and systems required for the defined functions. Always remove/disable debug and development features in devices when creating production code

The larger the body of code, the greater the chance of undiscovered security flaws. Therefore, it is prudent to remove as much code as possible to limit the chance that a flaw discovered after shipping the product will require patching or other mitigations. This includes code that may not be normally executed; even if the code is not used, having it within the device can lead to security problems down the road.

For similar reasons, it is essential to remove debug and development code from the device prior to shipping. This is often done with isolating "ifdef" statements, which can automatically remove such features during compile time. Although it is understandable to want features in the code in case there are problems in production, it is often more likely that such features will become a source of vulnerability as they provide access and information that would not normally be provided in the end-user environment.

20 Implement a vulnerability management program to regularly monitor and address security flaws in the product prior to release and through end-of-life. Include a process to distribute patches to customers and keep them informed

Security is a moving target, and no system will ever be 100% secure. As new vulnerabilities and attack methods are released, it is important to have a program to ensure that systems – both new and existing – are not vulnerable to these security flaws. This is only possible when there is a management-endorsed and enforced program to ensure ongoing monitoring and updates of system security.

The ability to install patches into systems is of no value if the patches are not created and made available for the customer systems to install.

To learn more about UL's Cybersecurity Services, visit UL.com/cybersecurity or email: ULCyber@ul.com

UL.com/cybersecurity

CT0119