VDOO

# VDOO Guideline

## ▶▶▶ 8 Best Practices for Developing Secure IoT Devices

## 8 Best Practices for Developing Secure IoT Devices

As the acronym implies, IoT devices are exposed to the local network and the web, making them susceptible to attacks by malicious actors. In the past few years, most high-profile attacks in the IoT space have been used to either achieve a political goal (such as with Stuxnet) or to achieve an illegal commercial goal (such as the Mirai botnet).

This post presents development and deployment recommendations for IoT device manufacturers to enhance baseline security. Following and implementing these practices requires relatively little effort but can significantly improve the level of security in the shipped device. It will not try to tackle or discuss advanced hardening measurements (such as anti-tampering and anti-debugging).

## 8 Best Practices for Developing Secure IoT Devices

Often, attackers need to compromise many devices to perpetrate their attacks. This is certainly true of DDoS attacks that rely on sending queries from hundreds of thousands of devices simultaneously. This is also true for crypto-mining. But whether the hacker wants to run a large, coordinated attack or attempt an individual breach, securing each device is of paramount importance.

To gain access, it is usually simplest for the attacker to target the lowest common denominator - the weakest points - across IoT devices, such as default passwords, known vulnerabilities, insecure and outdated libraries and applications, exploitable bugs, and similar security vulnerabilities.

## Secure Foundations

**1: Choose a well supported base distribution**

IoT device security is best served by using versions of Linux from a security-oriented base distribution, rather than a custom Linux distribution.

For organizations that have the resources, building a custom OS distribution might be an attractive option. This allows for an optimized bill of materials - because of lower memory, processing power and storage requirements - for a better user experience, for example, if the interface latency is lowered or if less network traffic is generated. From a competitive standpoint, it gives the vendor complete control of the pipeline and minimizes the reliance on external vendors to provide security fixes or advance the state of the art. Major corporations, such as Google, Apple, Amazon, Samsung, and LG maintain their own brand of OS with mixed success.

However, from a security perspective, that level of customization is unwise. When you customize an OS in that way, you lose all security oversight and open yourself up to possible exploitation.

Bug tracking, administering, modifying, and updating the environment to respond to the constant onslaught of new threats is an onerous task which requires thousands of man-hours each year to handle - this is unfeasible for most smaller organizations.

We strongly recommend relying on the efforts of the Linux community and choose a well-supported and security-oriented base distribution, such as Ubuntu Core, Debian, YoctoLinux, or Automotive Grade Linux.

## Secure Foundations

VDOO

**2: Implement a remote update mechanism for your device**

Choosing a well maintained base distribution is a great starting point. Over time, as new bugs and security vulnerabilities are found and fixed, and the overall quality of the base product improves, it is important to keep your devices up to date. Implementing a remote update mechanism enables the base distribution and your device application to receive security patches without explicit user or operator intervention.

The update process itself is safe, most of the updates use at least private/ public keys, package hash verification and secure transport (TLS).

The device type, software dependencies and storage strategy in use might require different approaches to the automatic updates. To avoid unintended consequences, check the updated libraries before pushing them to the end-user. Leave enough free space to compensate for limited write count on solid state storage. Ubuntu, for example, has the "unattended-upgrades" package that is installed by default. With some adjustments, it can keep your device up-to-date.

Other Linux distributions could have multiple update mechanisms. YoctoLinux, for example, has five different mechanisms, so careful research is required. See here.

## Secure Foundations

VDOO

**3: Use analysis tools to find security hardening recommendations**

Static analysis of source code is an essential step toward hardening your software. A variety of excellent solutions are available.

Some mainstream Linux distributions (for example, Raspbian) are geared towards user experience and ease of development. In these cases, security is often at a lower priority. Even so, it is possible to achieve good baseline security.

Security hardening can often be done by simply modifying configuration options. This can be done either automatically or by following published guidelines, such as those mentioned in the archlinux wiki and in the Linux Debian manual. The basic functionality can also be validated with automated tools.

Lynis by CISOfy mostly deals with server configuration but it can still be used to find some action points. For example, verifying that the SSH configuration is hardened and that the file permissions are set up rigorously. At the time of writing, Lynis has more than 380 different hardening tests.

Dev-Sec is also an excellent hardening test framework. Like Lynis, Dev-Sec is more geared toward server security.

Metasploit is a well-known network-oriented penetration testing framework. It requires some expertise to operate

## Secure Development – During Development

### 4: Use static analysis tools to eliminate trivial bugs

Trivial bugs, such as not closing file handles or unintended fall through in switch cases (for example CVE-2014-1266 - iOS SSL Exchange) can be used to crash your device or gain unauthorized access. A complete test coverage of the codebase is extremely time consuming, and in most cases, impossible to achieve.

Fortunately, there are multiple static analyzers available that can detect and prevent a wide range of these types of bugs. While not completely effort-free (false-positives are relatively common) the benefit-to-cost ratio is significant.

Cppcheck for C/C++ is a first-rate free tool for easy baseline security. Coverity and PVS-STUDIO are excellent commercial static analysis suites.

Coverity and PVS-STUDIO are excellent commercial static analysis suites

## Secure Development – During Testing

### 5: Use dynamic analysis to find memory bugs

In the early part of this century, most of the security breaches were due to memory handling errors - stack and heap overflows, page faults, and so on. Well-known examples are the Slammer and Blaster worms, which affected hundreds of thousands of machines.

Dynamic memory analysis - monitoring and capturing anomalous behavior during runtime - can be used to eliminate memory handling bugs. Several free tools are available that do a good job of this.

The leading free dynamic analysis tools include Address-sanitizer (ASAN) and  Valgrind.

## Secure Development – During Testing

VDOO

### 5: Use dynamic analysis to find memory bugs

Address-sanitizer (ASAN)

Originally a research project by Google, ASAN can detect memory leaks, allocating errors, and address space access errors (and more), both on the heap and the stack. ASAN requires linking against address sanitizer libraries and running the modified binaries. ASAN is supported by GCC 4.8 and up, and Clang 3.1 and up.

A major limitation of ASAN is that to detect heap use errors, ASAN overrides malloc and free, so using custom memory allocation libraries (such as jmalloc) will limit this functionality. Mozilla was able to use ASAN when developing Firefox by adding a compilation flag that redirects jmalloc calls to malloc and free.

The runtime impact, according to Google researchers, is 73% CPU and 340% for memory, so make sure that your hardware can handle it. Alternatively, write a cross-platform code that can run on more powerful hardware for testing purposes, and remember to disable it when the software is compiled for release.

Add the -fsanitize=address flag to your test build file, and you're set.

## Secure Development – During Testing

### 5: Use dynamic analysis to find memory bugs

<u>Valgrind</u>

Valgrind can detect the same problem classes as ASAN, but it can do much more. For example, it can simulate low memory, profile heap and cache use, and find race conditions. Valgrind is essentially a virtual machine, so it does not require changes to the binaries or the build process, which is good when using a custom memory allocation scheme.

On the other hand, Valgrind supports a much more limited range of hardware, and the CPU impact is up to 500% of the original load.

## Upon release

### 6: Remove Frame Pointers, enable Compiler Optimizations

The frame pointer is used as a reference to the location of variables of the stack. It is required on some architectures to allow locating functions and their arguments in memory during debug.

Removing Frame Pointers will make debugging much harder, if not impossible, on some architectures, and as a side effect, will make it far more difficult to reverse engineer your product.

Compiler Optimization results in leaner code that's much further from the original source than regular, unoptimized binaries. Compilers introduce numerous changes in optimized binaries, making optimized assembly much harder to read for a reverse-engineer.

On Linux platforms, the O3 optimization level removes the instructions to save, set up and restore Frame Pointers. This frees up an extra register and makes debugging impossible on some machines.

By default, GCC disables the omit Frame Pointer optimization when it interferes with debugging. We recommend enabling this function.

For good measure, raise the optimization to the maximum possible.

## Upon release

VDOO

### 6: Remove Frame Pointers, enable Compiler Optimizations

On Linux:

> pass -O3 -fomit-frame-pointer

On Windows, use:

> /Oy /Ox

For further information, read the following manual or here

## Upon release

### 7: Use compiler security flags

Many security vulnerabilities are caused by errors that are common and relatively easy to find. Well known attack vectors include memory leaks, buffer overflows (one was found in the recent Foscam vulnerability - CVE-2018-6832), unbounded inputs, allocation failures, and reading from non-secure sources. Finding all potential bugs is a difficult and potentially infinite task.

On many platforms, the compiler and target architecture support security features that can detect and prevent many of these issues by compiling the software with the hardening features enabled (there may be some performance impact, however).

If your compiler and target architecture support security features, becoming familiar with and enabling them is well worthwhile.

## Upon release

### 7: Use compiler security flags

For the GCC compiler for example, Hardened Gentoo recommends using the following flags:

> CFLAGS="-fPIE -fstack-protector-all -D_FORTIFY_SOURCE=2" LDFLAGS="-Wl,-z,now -Wl,-z,relro“

Let's explain some of those flags:

PIE enables the 'position-independent executable', part of Address Space Layout Randomization (ASLR). When attackers achieve write privileges to a memory location, they can use it to jump to some other memory location, such as a function that prints all the passwords on the device. To do so, the attacker needs to know the memory address of the target function to make the jump.

An early version of ASLR on the iPhone randomized between 128 different locations, so hackers wrote consecutively to all of the locations and achieved access.

## Upon release

VDOO

**7: Use compiler security flags**

For the GCC compiler for example, Hardened Gentoo recommends using the following flags:

> CFLAGS="-fPIE -fstack-protector-all -D_FORTIFY_SOURCE=2" LDFLAGS="-Wl,-z,now -Wl,-z,relro“

When modern PIE/ASLR is enabled, the memory layout of the program changes on each runtime, so even if attackers can arbitrarily jump anywhere in the program, they won't know where to aim.

-fstacker-protector-all enables stack protection on all functions. This protects against buffer overflows, adds guard variables (to prevent writing outside of the buffers boundaries), and more.

RELRO is Relocation Read-only and it prevents changing dynamically linked libraries after the application is launched. This prevents an attacker from linking to compromised dynamic libraries after the program is launched. The reported performance hit for using these types of security functions is about 10%. In most cases, this should not be a problem. In all cases, reduced performance is better than surrendering the device to a botnet.

For further reading, see: Hardened Gentoo, Hardening Debian, GCC security related flags reference.

## Upon release

VDOO

### 8: Strip binaries, compile in release mode

Debug symbols are extremely useful in the development phase when debugging, or after deployment, when a problem arises in the field and a better insight is needed to understand the problem.

For the same exact reason, debug symbols are also useful to reverse engineers when attempting to attack your device. Reverse engineering can be made harder by stripping the binaries and removing the debug symbols.

On Linux, Using GCC or LLVM/Clang

- Release mode
  - Make sure that the -g flag is not enabled in the build system
- Stripping
  - Strip the binaries by adding -s to the build system
  - Use the GNU/strip utility

## Conclusions and Warning – Security is Transient

Most existing IoT devices have low to very-low security standards, with the status on the IoT landscape being as bad as it was 30 years ago on the desktop.

Eventually, most high-quality IoT vendors will catch up to the best practices and the base state of the IoT landscape will improve. As an evolutionary reaction, the threat landscape will adapt as well and new classes of vulnerabilities will be found and utilized (such as the Meltdown branch prediction vulnerability), the bar for baseline security will rise accordingly, and the cycle is bound to repeat again indefinitely.

Keeping IoT devices secure is a commitment and a continuous effort. The bottom line is that following basic best practices will prevent your device from being among the lowest common denominator - the weakest link and the obvious target for most attackers.

Contact us to learn more about how the VDOO platform can help in integrating security into an IoT SDLC.

**Tomer Zaidenstein, VDOO**