NCC Group Whitepaper

# Improving Your Embedded Linux Security Posture with Yocto

August 27, 2018 – Version 1.0

Prepared by
Jon Szymaniak, Senior Security Consultant

Abstract
Embedded systems are regularly found to lack modern security-focused designs and implementations, despite decades of advancements in the field of computer security. Although the emergence and adoption of projects such as Yocto and OpenEmbedded have made it easier to develop and maintain firmware for embedded Linux systems, NCC Group has often found that engineering teams are not utilizing these tools to their full potential. As a result, security assessments yield numerous findings that could have been detected and remediated much earlier in the product life cycle. This whitepaper introduces functionality available within the Yocto ecosystem that can be leveraged to integrate security-focused QA into firmware build processes. By adopting the practices and guidelines presented in this paper, your team will be able to improve their baseline security posture and obtain more value from their investments in product security.

# Table of Contents

# 1 Introduction

## 1.1 Challenges Faced by Firmware Development Teams

Over the last two decades, Linux-based firmware has found a home in a breadth of application domains, including networking equipment, industrial imaging systems, healthcare devices, automotive infotainment systems, and consumer electronics, just to name a few. As anyone tasked with developing, maintaining, and releasing firmware for such products may know, managing modern firmware builds can become challenging and complicated due to numerous factors.

Despite the ever increasing complexity of system on a chip (SoC) solutions, engineering teams are often faced with a fixed or even decreasing desired time to market. In order to meet tight project deadlines, teams must often rely upon third-party software and libraries. Although it may have been easy to jumpstart a product using a silicon vendor's pre-built software development kit (SDK) or board support package (BSP), heavy reliance on these items can incur significant technical debt that must be accounted for later. Oftentimes silicon vendors do not release updates to these packages, leaving engineering teams with the responsibility of patching third-party software and navigating massive patch deltas when attempting to update core libraries or operating system components.

When dealing with complex architectures consisting of numerous software components, some organizations may find that an extremely small number of individuals actually possess the tooling and knowledge needed to produce a complete firmware image. As a result, few members in the organization may have a holistic understanding of the state of their product's firmware, as well as the origins of various software packages and configurations.

Whether it is due to complicated or fragmented processes, a lack of knowledge transfer, or perhaps fragile build environments relying upon outdated software, this situation can easily result in firmware that is difficult to reproduce, review, test, and audit. **Although these challenges may not appear to have immediate product security consequences, they help create conditions in which serious security vulnerabilities can go unnoticed until well after a product's launch.**

## 1.2 Tooling for Improved Embedded Linux Workflows

Fortunately, a handful of projects have been developed over the last decade to address the aforementioned challenges, including OpenEmbedded, the Yocto Project, and Buildroot [1–4]. These projects simplify the process of configuring cross-compilation environments, and obtaining, building, and deploying software for a variety of processor architectures. Since their inception, these projects have seen significant adoption rates, especially as silicon vendors and their third-party partners have begun leveraging them to provide reference designs and SDKs [5–9].

However, the adoption of these tools alone does not immediately guarantee a more robust product. In order to maximize the value obtained from these tools, organizations must first identify the weaknesses in their current workflows and then integrate new tooling in a manner that specifically addresses these weaknesses. Within the context of product security, this often involves first establishing a secure software development life cycle (SSDLC) and then identifying how available tools can be utilized to satisfy SSDLC objectives.

When performing security assessments of Linux-based embedded systems, NCC Group frequently observes the presence of known or commonplace security vulnerabilities, along with a lack of standard exploit mitigations. **In these situations, readily available functionality found within the Yocto ecosystem could have been leveraged to detect and address many of these issues.** Furthermore, increased adoption and coordinated use of the tools across teams might have better facilitated build and quality assurance (QA) automation, allowing issues to be detected and resolved earlier in the product life cycle, when the remediation of security defects incurs a significantly smaller cost impact [10].

## 1.3 Focus and Goals

This paper focuses on the Yocto Project, as of the version 2.5 ("Sumo") release. It is intended to help embedded systems engineers and QA teams make better use of available features in a manner that helps improve their product's overall security posture. In particular, this paper aims to offer guidance on how to:

- Improve traceability and team members' understanding of firmware contents (Section 2)
- Customize and automate QA procedures to better meet organizations' SSDLC objectives (Section 3)
- Better adhere to quality and security best-practices (Section 4)
- Prepare materials that can be used to yield more productive security assessments (Section 5)

The recommendations discussed in this paper are intended to be followed in the order in which they are presented. For example, attempting to harden and patch application software before identifying and eliminating extraneous functionality could lead to wasted development effort. Therefore, functionality presented in Section 2 is important to pursue before following recommendations discussed in Section 4.

More advanced security topics are not covered in this paper, as this **is not** intended to be a comprehensive Linux hardening guide. Instead, this document aims to help readers improve firmware build processes in a manner that integrates and enforces security fundamentals. By following the guidelines presented in this paper, readers will find they are better equipped to detect and eliminate common security defects. As a result, time and effort can instead be focused on more comprehensive testing and review of in-house application code, mitigating risks associated with third-party components, and integrating more advanced security functionality.

Lastly, this paper does not serve as a replacement for the thorough documentation maintained by the Yocto Project [11] and the OpenEmbedded community [12]. Instead, the purpose of this document is to guide the reader to specific functionality outlined in the official documentation, and then offer insight into how to best utilize it to achieve the aforementioned goals.

## 1.4 Intended Audience

This paper assumes that the reader already has some familiarity with the Yocto Project and is comfortable with its terminology, writing BitBake recipes, and building firmware images. Yocto and OpenEmbedded variables are frequently referenced, so having the Yocto Project Reference Manual [13] open to the *Variables Glossary* section is highly recommended.

No formal background in computer or embedded system security is assumed. This paper introduces a number of fundamental security concepts and provides references to supplemental materials.

For readers who are new to the Yocto Project, it is recommended that they first review either the official quick build guide [14] or the "*Ready, Set, Yocto!*" tutorial [15]. It is helpful to follow along with these guides using a low-cost platform, such as a Raspberry Pi Zero [16].

# 2 Improving Traceability

When working with clients to perform white-box security assessments of their Linux-based embedded platforms, some of the items NCC Group often requests are architectural diagrams, software design documentation, and a list of software components installed on the production systems, along with their associated version numbers. This information allows NCC Group to quickly begin assessing the platform's overall attack surface[1] and establish a test plan for the engagement.

In some cases, the exercise of attempting to obtain or consolidate this information can be rather eye-opening to a client. These design artifacts may not have been updated to accurately reflect the implementation after requirements changed or after designing work-arounds for unforeseen challenges. In situations where multiple teams are contributing different components to the firmware, there may not be a centralized record of which components were introduced or otherwise required by another team.

Without a clear picture of which build artifacts are included to satisfy product requirements and which are vestiges of development efforts, diagnostic software and interfaces can remain unintentionally deployed and enabled in production firmware builds. Oftentimes, these items are included as dependencies of production components, making it more difficult to track down the reason for their inclusion. Such items represent the low hanging fruit for attackers because they commonly provide privileged access to sensitive resources with few to no security barriers.

Without a clear understanding of what software components are deployed on a system, it can be difficult to plan and execute a thorough risk assessment, and it can become extremely tedious to determine which security advisories affect the various firmware releases of a given product.

**It is critical that an organization documents and maintains an understanding of exactly what software is deployed in their product, why it is required, and which team (member) is responsible for it.** This section highlights a few features and methods by which teams can gain an understanding of the state of their firmware.

## 2.1 Standardizing a Version Control Workflow

Building firmware using Yocto and OpenEmbedded largely consists of creating and editing metadata files to describe the desired firmware contents. Therefore, in order to maximize your visibility into how and why firmware images have changed, modifications of this metadata **must** be tracked in a consistent and disciplined manner.

Modern version control systems are specifically designed to satisfy this need. Because the Yocto project itself uses `git` [17], this is naturally a good fit for tracking your own layers. However, the version control system you decide to use is ultimately a matter of what best fits your organization. It is more important that the version control workflow you adopt is applied consistently and in a manner that provides value when seeking to identify the change(s) responsible for a new defect or security vulnerability. This consistency becomes even more crucial if teams are geographically isolated; the version control history can be used to understand the rationale for changes at times when their associated authors are not available.

Disciplined use of version control will likely require an investment in your employees' education, and a continued maintenance of your own organization's guidelines. Fortunately, helpful resources are now plentiful online [18–21]. Appendix A presents a few high-level guidelines to consider when establishing a workflow across your organization.

## 2.2 Leveraging Yocto's Build History

Yocto's "Build History" feature [22] provides much of what is necessary to gain insight into both the current state of a firmware build, as well as how it has changed over time. This section discusses the types of information made available by the build history feature, how to configure its various settings, and how to effectively use the information it provides.

[1]The "attack surface" of a system refers to all of the different entry points an attacker can try to leverage to gain unauthorized access. On embedded systems, this not only includes the hardware and software interfaces exposed to the outside world (e.g. USB, Ethernet), but also internal signals and buses between components and modules. Reducing a system's overall attack surface is a fundamental security best practice, which is reiterated throughout this paper.

The type of information included in the build history output is configurable via a `${BUILDHISTORY_FEATURES}` definition in your `local.conf` file. If not explicitly defined, this variable defaults to `"image package sdk"`, which provides a significant amount of information about images, packages, and generated SDKs.

The generation of build history output increases both build times and disk space utilization. Therefore, it may be best to start by enabling this feature at its most basic level, and gradually enable additional information as needed. To produce information only for images, add the snippet shown in Listing 1 to your `local.conf` file and re-run a build of your top-level image.

```
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "0"
BUILDHISTORY_FEATURES = "image"
```

**Listing 1:** Enabling image build history information only

After the build completes, a "`buildhistory`" directory will have been created in the build directory, with a number of text files written to the location shown in Listing 2. Note that this listing presents the directory location in both a symbolic form and an expanded form, using example values for the variables. Colors are used to indicate the association between a variable and its expanded form.

```
${BUILDHISTORY_DIR}/images/${MACHINE_ARCH}/${TCLIBC}/${IMAGE_BASENAME}
/home/demo/builds/buildhistory/images/raspberrypi0_wifi/glibc/ncc-demo-image
```

**Listing 2:** Image-specific build history directory

### Image Contents

The following files, found in this directory, are particularly useful for understanding exactly what is deployed in an image. **When preparing a firmware release, perform a review of these items, and be sure to archive these build artifacts for future reference.** As you update third-party layers in the future (e.g. to integrate upstream security patches), this information will be critical in determining which of your previous firmware versions may be affected by a particular security advisory.

| Filename | Description |
|---|---|
| `build-id.txt` | This file captures the state of the system and BitBake layers at the time the image was created. Note the presence of the version control information associated with each layer included by your `bblayers.conf` file. |
| `files-in-image.txt` | All deployed files and their permissions are included in this file. |
| `installed-packages.txt` | All software packages and their version information can be found in this file. |

**Table 1:** Image-specific build history artifacts

Because `installed-packages.txt` includes additional information, such as recipe revision and target architecture, you may wish to strip this supplemental information before processing it in a script or including it in a build manifest. The following snippet illustrates one way to do this via `sed` and a regular expression.

```
sed -e 's/-r[0-9]\+\..*//' installed-packages.txt
```

**Listing 3:** Removal of additional package version information with `sed`

## Image dependencies

The `bitbake -g` option allows recipe dependencies to be exported to `dot` files.[2] When used in conjunction with the `-u taskexp` option, dependencies can be reviewed on a per-task basis in the Task Dependency Explorer GUI.

However, if the reason for a software component's inclusion in an image is unknown, attempting to view dependencies for an entire image to identify the relevant dependency relationships can be cumbersome. The `dep-subgraph.py` Python script presented in Appendix C[3] exemplifies one simple way in which a subset of a build history image dependency graph can be extracted.

Listing 4 demonstrates this script being used to visualize the subset of packages in the `core-image-sato` image[4] that depend upon `libgcrypt`. Line 1 invokes the `dep-subgraph.py` graph, specifying that four levels of dependency information should be included. Line two invokes the `xdot` program, allowing the graph to be interactively viewed. Finally, the subgraph is converted to an SVG image file on line 3. The resulting image is shown in Figure 1. Solid lines represent build and runtime dependencies corresponding to `DEPENDS` and `RDEPENDS` recipe entries, respectively. Dotted lines indicate the dependencies resulting from the use of `RRECOMMENDS`.

```
1  $ ./dep-subgraph.py -i ${BUILDHIST}/depends.dot -o deps.dot -p libgcrypt -H 4
2  $ xdot ./deps.dot
3  $ dot -Tsvg deps.dot -o deps.svg
```

**Listing 4:** Extracting and viewing packages depending upon `libgcrypt`

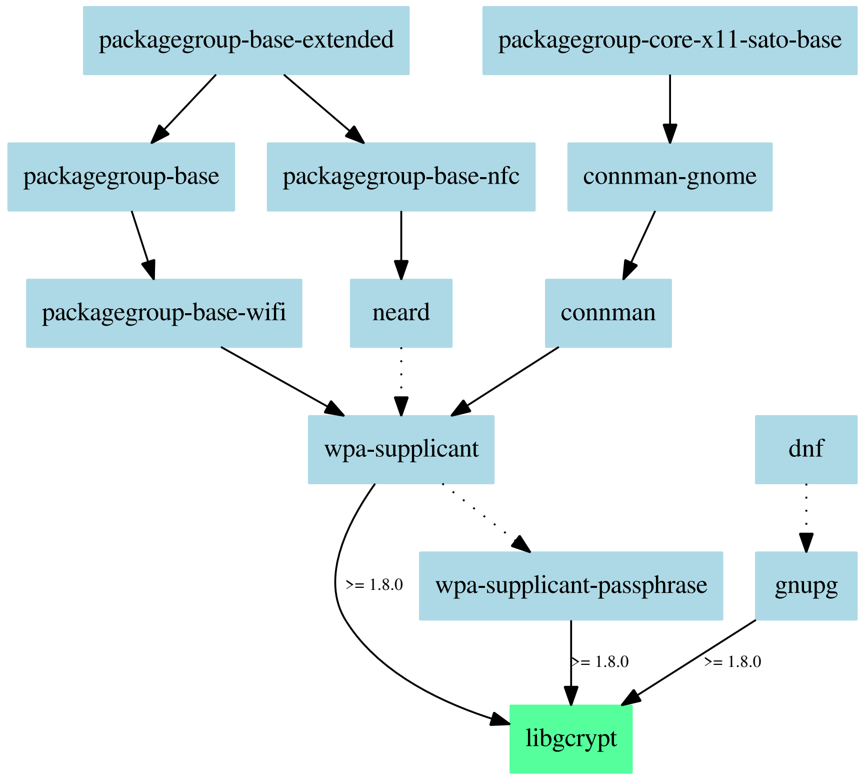

**Figure 1:** Packages depending upon `libgcrypt` in `core-sato-image`

[2]See [23,24] for more information about the DOT file syntax and associated Graphviz software.
[3]Available online: https://github.com/nccgroup/yocto-whitepaper-examples
[4]This image, included in Poky releases, includes a graphical X11 environment intended for older mobile devices. It is used here as an example because it deploys a larger number of packages than other readily available examples, such as `core-image-minimal`.

## Package Information

Adding "package" to the definition of BUILDHISTORY_FEATURES introduces additional per-package information in the build history output. This not only consists of recipes' variable definitions for each exported package (e.g. DEPENDS), but also provides a complete listing of files deployed in each package. The path to this information can be found in Listing 5.

```
${BUILDHISTORY_DIR}/packages/<architecture>/<recipe>/<package>/files-in-package.txt
```

**Listing 5:** Path to files-in-package.txt

Similar to the image build history's files-in-image.txt, this text file contains files' locations, sizes, and permissions. These build history files can be helpful when seeking to establish the root cause of a failure or security-impacting defect, particularly when it is the result of an incorrect or unintentional deployment of a build artifact. The files-in-package.txt files can be used to establish the association between a particular file and the package created by a specific recipe. The image dependency information discussed in the previous section can then be used to review the dependency chain that resulted in the deployment of the particular build artifact.

Consider a situation in which a platform is found to initialize an attached USB Ethernet interface and obtain an IP address via DHCP, resulting in the exposure of a diagnostic service. After finding that the /etc/network/interfaces file deployed in the root filesystem is responsible for this behavior, one could locate the recipe deploying this file as follows:

```
$ cd ${BUILDDIR}
$ find ./buildhistory -name 'files-in-package.txt' \
                      -exec grep -l '/etc/network/interfaces' {} \;

buildhistory/packages/arm1176jzfshf-vfp-poky-linux-gnueabi/init-ifupdown/init-ifupdown/      ↩
files-in-package.txt
```

**Listing 6:** Locating a build artifact using per-package build history

The above output indicates that the init-ifupdown recipe provides an init-ifupdown package containing this file. One way to further determine the location of this recipe, along with any bbappend [25] files that may be altering its default behavior, is to inspect the log files generated during its tasks' execution. For example, the following log file includes the search paths used to locate files in the recipe's ${SRC_URI} definition, which should include the location of any relevant bbappend files.

```
${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/init-ifupdown/1.0-r7/temp/log.do_fetch
```

**Listing 7:** Example log.do_fetch file location for the init-ifupdown recipe

If it is suspected that a bbappend file, or even a conflicting recipe from another layer, may be the source of a problem, the bitbake-layers command can be used to quickly identify these items. See the help text for the bitbake-layers show-appends and show-overlaid subcommands for more information.

## Build History Logs

Using the Build History feature with ${BUILDHISTORY_COMMIT} set to "1" results in every change to the build being tracked in a git repository stored in ${BUILDDIR}/buildhistory [26]. While version control of your custom layer(s) helps capture the recipe-level changes and their intent, this feature provides additional insight into the lower-level build changes in terms of build artifacts.

This feature will likely provide the most value when used on continuous integration systems responsible for nightly development builds or product-intent release builds. Because it consumes additional build time and disk space, it may not be practical to use on individual developers' machines.

Although it is certainly possible to review the changes via `git`, the `buildhistory-diff` command tends to be more palatable, as it presents build changes in terms of added and removed files, package changes, and in some cases, the specific changes made to deployed files [27]. Be aware that changes to deployed files that are not otherwise associated with recipe or package-level changes may not be reported. Instead, only a package size difference may be logged.

For example, a custom `interfaces` file is newly introduced in a root filesystem via the use of the `bbappend` file shown in Listing 8. Re-running the build of an image only results in a size difference being reported for the `interfaces` file in `files-in-image.txt` and `files-in-package.txt`. Invoking `buildhistory-diff` does not output a diff of this file, which may mislead one to believe that no change has occurred.

The deltas introduced by the custom `interfaces` file should be captured in the associated layer's version control history, so these changes are not totally untracked. However, if the lack of reporting in this particular build history example is problematic, consider augmenting the build history class to track the hashes of files deployed in images.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

**Listing 8:** An `init-ifupdown_1.0.bbappend` snippet used to deploy a custom `interfaces` file from `files/`

Prior to firmware releases, review both the version control history for layers and this build history information to determine if the metadata-level changes (i.e. to layers and their recipes) account for all the resulting build artifact changes. Treat unaccounted for items as potential security risks until confirmed otherwise. If the volume of changes between releases renders this task impractical, more frequent QA reviews might be beneficial.

# 3 Automating QA Tasks

While it is certainly valuable to aggregate and store plenty of information about firmware builds for future reference, manually working with this information can be tedious and error-prone. By leveraging Yocto's ability to integrate QA tasks into the build processes, engineering teams can quickly become aware of issues before builds are delivered to QA and testing teams. This ultimately allows testing teams to remain focused on triaging defects and performing more comprehensive reviews.

## 3.1 CVE Database Queries

As of the Yocto 2.2 Morty release, a `cve-check` BitBake class [28] is included in the `poky/meta/classes` directory [29]. This class may be used to supplement[5] existing processes for monitoring security advisories.

This class operates by downloading and maintaining an updated copy of the National Vulnerability Database (NVD) [30]. By adding the following line to your custom distribution configuration (discussed in Section 4.2.1) or `local.conf` file, a `do_cve_check()` task is added to each recipe.

```
INHERIT += cve-check
```

**Listing 9:** Inheriting from the `cve-check` BitBake class in a `local.conf` file

During a build, the local copy of the NVD is queried for each recipe included in the build. If a recipe's base package name (`${BPN}`) and package version (`${PV}`) are associated with one or more Common Vulnerabilities and Exposures (CVE) Identifiers [31], this information is logged and a build warning is printed. An example of this is shown in the following snippet, in which `wpa-supplicant 2.6` is reported to be vulnerable to the KRACK family of vulnerabilities [32].

```
WARNING: wpa-supplicant-2.6-r0 do_cve_check: Found unpatched CVE (CVE-2017-13077     ↩
CVE-2017-13078 CVE-2017-13079 CVE-2017-13080 CVE-2017-13081 CVE-2017-13082 CVE-2017-13084   ↩
CVE-2017-13086 CVE-2017-13087 CVE-2017-13088), for more  information check     ↩
${BUILDDIR}/tmp/work/arm1176jzfshf-vfp-poky-linux-gnueabi/wpa-supplicant/2.6-r0/cve/cve.log
```

**Listing 10:** Example `cve-check` output

As shown in Listing 11, details are presented in a simple `KEY:VALUE` format that can easily be parsed by additional scripts. When building an image, this information is consolidated into a single text file for all recipes in the image and stored alongside the image in the deployment directory, with a `.cve` extension. This build artifact could then be ingested by other scripts, such as those run by post build steps on a continuous integration (CI) server, to open and assign issue tracker items.

```
PACKAGE NAME: wpa-supplicant
PACKAGE VERSION: 2.6
CVE: CVE-2017-13077
CVE STATUS: Unpatched
CVE SUMMARY: Wi-Fi Protected Access (WPA and WPA2) allows reinstallation of the Pairwise   ↩
Transient Key (PTK) Temporal Key (TK) during the four-way handshake, allowing an attacker   ↩
within radio range to replay, decrypt, or spoof frames.
CVSS v2 BASE SCORE: 5.4
VECTOR: ADJACENT_NETWORK
MORE INFORMATION: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2017-13077
```

**Listing 11:** A snippet from the `wpa-supplicant 2.6 cve.log` file

When leveraging this feature to monitor production builds, it is important to ensure that recipe names and their version number formats properly match the information maintained by the NVD. Otherwise, relevant CVEs may not be reported.

[5] This class uses the National Vulnerability Database as a single source of information. Therefore, it is critical that it is not relied upon as the *only* source of security advisory information in your organization.

In the event that a recipe contained in a third-party layer specifies a `${BPN}` or `${PV}` definition that is inconsistent with the corresponding NVD entry, this information can be overridden in a `bbappend` file using the `${CVE_PRODUCT}` and `${CVE_VERSION}` variables.[6]

False positives are inevitable when using vulnerability databases – the vulnerability may not be relevant for your system's configuration and use-cases, or the high-level database entry may not capture the subtleties of when the defect actually presents any risk. If too many of these false positives are present, valid warnings can become lost in the noise; the CVE checks result in more harm than good.

Therefore, to obtain the most value from the `cve-check` functionality, it is important to prune false positive warnings. This should be done in a way that captures the reason why a CVE is not applicable, such that team members can determine if future changes introduce new risks. Adding this information to a version-controlled layer (Section 2.1) should help achieve the desired traceability.

A `${CVE_CHECK_CVE_WHITELIST}` variable is defined in `cve-check.bbclass`. This allows specific CVEs to be ignored for certain package versions. One way to suppress false positives is to create recipe-specific `bbappend` files in your organization's Yocto layer(s) and override the definition of this variable.

For example, when using Poky's `rocko` release branch, commit `de57fd8` introduces `wpa-supplicant` fixes for the CVEs shown in Listing 10. However, with these fixes applied, `cve-check` still produces a warning indicating that CVE-2017-13084 is unpatched. This is a false positive for `wpa-supplicant_2.6.bb`; the NVD database information does not capture the specific details about the non-functional state of WPA Supplicant's PeerKey implementation in this version. Listing 12 presents an example `wpa-supplicant_2.6.bbappend` file that can be used to suppress this warning.

```
# CVE-2017-13084 does not affect wpa-supplicant 2.6 because the affected PeerKey
# implementation is not fully functional, and therefore poses no practical risk.
#
# See the "PeerKey / TDLS PeerKey" section of:
# https://w1.fi/security/2017-1/wpa-packet-number-reuse-with-replayed-messages.txt
#
CVE_CHECK_CVE_WHITELIST = "{\
    'CVE-2017-13084': ('2.6'), \
}"
```

**Listing 12:** Suppressing a false-positive CVE warning via a wpa-supplicant_2.6.bbappend file

Be sure to make note of the syntax used in this variable definition. `${CVE_CHECK_CVE_WHITELIST}` is actually a Python dictionary in which a CVE identifier string maps to a tuple of version number strings. After making this change and re-running the build, the warning should no longer be present and the CVE will no longer be included in the `.cve` file build artifact. If `bitbake` is invoked with the `-v, --verbose` option, the following message is also displayed:

```
NOTE: wpa-supplicant-2.6-r0 do_cve_check: wpa_supplicant-2.6 has been whitelisted for    ↩
CVE-2017-13084
```

**Listing 13:** A CVE whitelist note printed to console as a result of the `CVE_CHECK_CVE_WHITELIST` change

Another option for addressing false positives is to use `bbappend` files to add dummy patches to a recipe's `${SRC_URI}` definition. These dummy `.patch` files should contain the CVE identifier (e.g. `CVE-2017-13084`) in either the patch file name or in the body of the patch description, but no actual patch delta content.

One major difference between these two methods is that the former will not include the CVE at all in the final `.cve` text file build artifact, whereas the latter will list it as "Patched."

[6] `${CVE_VERSION}` was introduced in the Yocto Project 2.5 ("Sumo") release. It is not present in earlier releases.

**Important Note**: A patch to resolve `cve-check` false positives caused by a parsing defect has been merged into the upstream git repositories for OpenEmbedded Core (commit `8fb70ce`) and Yocto Poky (commit `dd5bf3e4`). This will be available in future Yocto Project releases, starting at version 2.6 ("Thud"). In the meantime, this patch, contributed by the author of this paper, can be obtained here:

https://git.yoctoproject.org/cgit/cgit.cgi/poky/commit/meta/classes/cve-check.bbclass?id=dd5bf3e4d2ad66d8572d622a147cea1a8fddb40d

## 3.2 Scanning Build Artifacts

Build-time scans of configuration files, applications, and filesystem images can help identify configuration and permissions issues that present security risks. The `meta-security` [33] and `meta-security-isafw` [34] layers provide tools that can identify and report these types of risks either at build-time, or on the target platform at runtime.

There is significant overlap between tools included in these two layers, which were developed separately despite having similar names. As such, it is prudent to first evaluate each tool independently to understand its strengths and limitations, and then determine which tools provide the most value to your organization. The degree to which the different tools included in these layers are actively maintained varies.

You will likely find that custom tooling will need to be developed to suit your needs, with the tools described in this section serving as baseline references or starting points. Ensure that the adoption of any one of these third-party tools does not become another technical debt; factor in the effort required to further develop and maintain such tooling in the long-term when evaluating your options.

### 3.2.1 meta-security

At the time of writing, `meta-security` is actively maintained and being updated with each Yocto release. It contains support for a breadth of security-related software, not all of which is relevant to this discussion. A subset of its contents are summarized here.

#### Build-time scans via buck-security

`buck-security` [35,36] is a security scanner that iterates over files within a root filesystem and performs a handful of basic checks associated with permissions and setuid/setgid bits. Although this tool no longer appears to be maintained,[7] it can certainly still be used as a reference for developing your own automated QA tests to run on generated root filesystems. It is described here to outline some useful checks to include in your own post-build QA BitBake classes (discussed in Section 3.3). This can be enabled by adding `meta-security` to one's `bblayers.conf` and then appending the lines shown in Listing 14 to an image's recipe.

```
DEPENDS += "buck-security-native"
inherit check_security
```

**Listing 14:** Enabling the `check_security` task in an image

When `check_security` is inherited, `meta-security`'s `check_security` BitBake class appends a `check_security()` function to the definition of `${ROOTFS_POSTPROCESS_COMMAND}`. This variable stores a list of functions that are invoked following the creation of an image's root filesystem [37]. After the root filesystem is staged, the `check_security()` function runs `buck-security` over its contents on the build host.

After building an image, the output of `buck-security` can be found alongside the recipe's log files in `${WORKDIR}`/temp, in a `log.do_checksecurity.${PID}` file. Appendix D contains an example log generated by a `core-image-sato` build.

The first check in the log, `emptypasswd`, identifies that no password is set for the root user; anyone with physical access

[7] The last commit to the `git` repository was in October of 2015.

to serial consoles associated with a TTY could log into the device as this user. This can happen if a `local.conf` file is left in its default state because `EXTRA_IMAGE_FEATURES` is initialized to contain the “`debug-tweaks`” option [38].

The `sgids` and `suids` checks report any binaries with the “set group ID” and “set user ID” bits set. In a multi-user system, these allow a user to execute an application with the privileges of another user or group [39]. If such an application contains a security vulnerability, an attacker may be able to leverage it to escalate their level of privilege on a system. As such, `setgid` and `setuid` functionality should be avoided in favor of other technologies[8] such as capabilities [42], seccomp filters [43], and Linux Security Module (LSM) implementations (e.g. AppArmor, SELinux) [44].

World-writable files and directories are reported by the last two checks in the log. Such files and directories may be indicators of weak access controls and poor separation between users on the system. Depending upon how these files and directories are used, an attacker may be able to abuse them in a manner that allows the attacker to escalate their privileges on a system. In this particular case, only the `/tmp` directory is reported. This is typical of many Linux distributions, but may not be strictly required by the software deployed in your firmware.

#### Runtime Scanners

Other scanners included in the `meta-security` layer, listed below, must be executed on the target platform. Most of these are general-purpose tools targeting enterprise server and desktop applications – none are specifically tailored to embedded systems. As a result, many of the items they scan and report may not be applicable for your product’s operating environment, use-cases, target architecture, and resource constraints. The scan results from these tools will require false-positive pruning after baseline scans. However, there may be value in evaluating their capabilities and methodologies when integrating security checks into regularly occurring QA testing.

Also note that many of these scanners require more full-featured shells or scripting environments than what is typically required on an embedded platform. To avoid polluting a production firmware image with these dependencies, consider using a separate image that extends the product-intent firmware. (Of course, this will require the differences in these two configurations to be enumerated and assessed.)

- `buck-security` - In addition to being run at build-time, this can also be installed on the target platform, allowing additional checks (e.g. firewall, sshd configuration) to be utilized.
- `checksec.sh` - This bash script identifies security features that a binary has been compiled with. (These features are further discussed in Appendix B.) As of 2011, this script is no longer maintained. A different author has begun maintaining an expanded version of this script on GitHub [45] that introduces additional checks, such as the presence of functions hardened by `FORTIFY_SOURCE`.
- checksecurity - The `checksecurity` Perl script [46] and its plugins are designed to periodically report the state of setuid binaries, passwordless superusers, changes in sockets, and iptable logs.
- `Lynis` - The `Lynis` [47] security auditing tool evaluates the software configurations and presents suggestions for addressing potential vulnerabilities. This tool is intended for enterprise compliance use-cases, but can be extended.
- `OpenSCAP` - The `OpenScap` tools [48] are also focused on auditing and managing standard enterprise security policies.

### 3.2.2 meta-security-isafw

The `meta-security-isafw` layer invokes the Image Security Analyser Framework (ISA FW) [49] via an `isafw` BitBake class. This provides additional security checks beyond what is offered by `buck-security`, covering both build configurations and filesystem contents. The ISA FW “plugins” discussed in this section are enabled by adding the `meta-security-isafw` layer to your `bblayers.conf` and appending the following line to your `local.conf`.

```
INHERIT += "isafw"
```

**Listing 15:** Inheriting the `isafw` BitBake class in a `local.conf` file

[8] Because these technologies are also utilized in Linux containers, [40] and [41] may also serve as helpful references.

All of the ISA FW plugins discussed below are executed following the completion of a build. The CVE Checker and License Analyzer may also be invoked without performing a build by running:

```
bitbake -c analyse_sources_all
```

**Listing 16:** Manual invocation of the `do_analyse_sources_all` task from the command line

The ISA FW plugins write their results to `${BUILDDIR}/tmp/log/isafw-report_${DATETIME}` directories, in both flat text and XML files.

**`ISA_cfa_plugin` - Compile Flag Analyzer**

The compiler flag analyzer uses `checksec.sh` [50], `execstack` [51], `readelf` [52], and `objdump` [53] to identify software components lacking standard exploit mitigation features. If you are unfamiliar with compilers' exploit mitigation features, refer to Appendix B before continuing. This appendix introduces concepts that are required to fully understand this section, as well as Section 4.3.

The CFA plugin also seeks to identify binaries that invoke `setuid()` and `setgid()` without calling `setgroups()` or `initgroups()`. This set of conditions may indicate a potential security risk if a binary does not correctly relinquish group privileges when dropping privileges.

Appendix E presents an abridged report from the Compile Flag Analyzer after building `core-image-sato` with largely default settings. Without the use of `security_flags.inc` (discussed in Section 4.3), this report will contain a significant number of items.

Note that both valid results and false positives are included in this report. It is important to understand which results are meaningful and why the false positives do not introduce any additional risk.

In the "Relocation Read-Only" section, both kernel modules and shared libraries are reported. The RELRO checks performed by the `checksec` tool only apply to items using the user space dynamic loader; the warnings for kernel modules can be ignored. Similarly, it is expected that shared libraries will be reported to have Partial RELRO, rather than Full RELRO enabled; the application that loads a shared library is ultimately responsible for remapping the Global Offset Table as read-only after symbols have been loaded and resolved. Applications listed as having no RELRO or partial RELRO are the items that can present legitimate risks. In this particular example, the `dbus-daemon-launch-helper` and `dropbear` applications have been compiled without Full RELRO protections, and therefore may present a greater risk if they were to contain a security vulnerability. Be aware that `checksec` falsely reports that statically compiled binaries have Partial RELRO protections; this is actually not the case, and statically compiled binaries are still susceptible to exploitation methods that aim to overwrite PLT and GOT sections [54].

Items under the "Stack protection" heading include applications, shared libraries, and kernel modules. The `checksec` tool should correctly report the presence of stack canaries in these items. Because the version of GCC used to produce this build uses a non-executable stack by default, no items will be reported as having an executable stack unless their build scripts explicitly disable this. (Such items should be reviewed to determine why this is the case.)

Under the "Grop initialization [sic]" heading, setuid and setgid binaries that do not contain references to `initgroups()` or `setgroups()` symbols are noted as potential risks. Further review of their ownership properties and how they relinquish their elevated privileges would be required to determine whether any items truly present any risks.

In the "Position Independent Executable" (PIE) section, dynamic shared libraries are listed as lacking PIE.[9] By nature, shared libraries must be position-independent to allow them to be dynamically loaded; this is not a concern. However, we again see that the `dbus-daemon-launch-helper` and `dropbear` applications have been compiled without this protection. As a result, they will not be able to leverage address space layout randomization, an important exploit mitigation.

[9] The `checksec` script outputs "DSO" for these dynamic shared objects, instead of PIE.

On non-x86 architectures, the "Memory Protection Extensions" items can be ignored. Otherwise, this section lists user and kernel-mode binaries compiled without the use of Intel-specific memory protection features.

### `ISA_cve_plugin` - CVE Checker

The CVE checker plugin searches for recipes affected by publicly disclosed vulnerabilities associated with CVE identifiers. This uses the same underlying `cve-check-tool` as the `cve-check` BitBake class included with Yocto Poky, so there is little to no additional value in using this plugin.

### `ISA_fsa_plugin` - Filesystem Analyzer

The filesystem analyzer is similar in functionality to that offered by build-time `buck-security` scans. It scans a generated filesystem and reports set-UID and set-GID binaries, world-writable files, and world-writable directories without a sticky bit set.

An example report, based upon a default `core-image-sato` build, is shown in Appendix F. This report notes the presence of a handful of setuid binaries and indicates that no world-writable or directory sticky-bit issues are present in the root filesystem.

### `ISA_kca_plugin` - Kernel Configuration Analyzer

This plugin parses the `.config` file used to configure the Linux kernel and searches for a fixed set of security-related settings that could potentially be improved. Checked settings include:

- Common diagnostic and debugging functionality that can be leveraged by an attacker
- Features that may unnecessarily increase the device's attack surface or disclose information to an attacker
- Whether or not any Linux Security Modules are enabled (e.g. SMACK, SELinux, AppArmor, tomoyo, Yama)
- Cryptographic key management functionality
- Kernel-level exploit mitigations

Be aware that configuration items are added, renamed, or removed as the Linux kernel changes over time. The ISA FW scripts represent a snapshot in time that do not include new features or renamed configurations. Furthermore, certain configuration options and recommended settings may only be applicable for certain architectures or platforms equipped with sufficient resources.

Snippets from an example KCA report are presented in Appendix G. This is indicative of a kernel in a development configuration that is not ready for production usage.

### `ISA_la_plugin` - License Analyzer

The license analyzer assists with auditing software licenses of (third-party) components deployed in an image. This illustrates how the build system could be used to ensure that software licenses are appropriately cataloged and that they do not conflict with your organization's own licenses and policies.

Although this is not necessarily a software security topic, it is mentioned here due to the source disclosure obligations associated with many Free and Open Source Software (FOSS) licenses. When a product contains both FOSS and proprietary software components, it is extremely important to ensure that the two are properly decoupled from one another, and that your legal team is involved in the associated review processes.

The `meta-security-isafw` layer contains `approved-non-osi`, `exceptions`, `licenses`, and `violations` files in its `isaplugins/configs/la` directory that configure which licenses are allowed or disallowed. An image may be whitelisted for these checks by adding it to a `${ISAFW_LA_PLUGIN_WHITELIST}` definition in your `local.conf`. After a build of an image, "problematic" licenses are written to a `la_problems_report_*` file in the ISA FW log directory.

Note that line 226 must be removed from `ISA_la_plugin.py`, or else the script removes the list of "unwanted" licenses directly after closing it. This appears to be a logical defect in the script introduced in commit `aa01cc6`.

```
os.remove(self.report_name + "_unwanted")
```

**Listing 17:** Line responsible for removing generated "unwanted" license report

As noted earlier, this plugin is best utilized as a reference for how the build system can be used to collect license information. Ultimately, a custom implementation will likely be required to fulfill your organizations' specific license auditing requirements. Some features to consider may include per-image reports and build-level warnings, as these can help development teams quickly identify problems as they arise.

**Limitations**

At the time of writing, the last update to `meta-security-isafw` occurred on June 13, 2017. Branches for currently supported Yocto releases (e.g. rocko, sumo) have not been added to the repository, but the `master` branch has been confirmed to work in conjunction with the rocko and sumo branches.

The infrequent updates might indicate that this project is not actively maintained. Plan accordingly if deciding to adopt this tool; your organization may need to maintain fixes and integrate features to support your workflow.

The KCA plugin currently includes architecture-specific items for x86 and arm. On other architectures, only the "common" checks are performed.

As previously noted, the `checksec` script (used by the CFA plugin) is no longer maintained. The expanded version [45] being maintained by another author reports additional items not covered by the CFA plugin, such as the lack of libc functions hardened via `FORTIFY_SOURCE` and binaries without their symbols stripped.

## 3.3 Adding Custom QA Tasks

Custom BitBake classes can be used to greatly extend the functionality included in Yocto's Poky releases. These can be leveraged to perform a variety of automated per-recipe and per-image QA tasks that are tailored to your products' or organization's specific needs.

After completing a security review of a product and then tracing security-impacting defects back to their root causes, custom QA tasks can often be introduced into build processes to detect regressions or similar vulnerabilities in future releases. Because these QA tasks are executed at build time, this is of course limited to defects that are quickly and statically observable. Long-running analyses, emulation, and dynamic testing are generally more appropriate for other post-build portions of a continuous integration system.

For example, on a number of occasions NCC Group has observed the unintentional deployment of sensitive files and highly privileged diagnostic application code in firmware releases. This section introduces two example BitBake classes intended to prevent these types of security risks through the use file and symbol blacklists. Refer to [12] for more information pertaining to the development of BitBake classes and tasks. The QA tasks implemented by `insane.bbclass` [55] may also serve as helpful examples.

### 3.3.1 Per-Image File Blacklist

Appendix H presents a `rootfs-file-blacklist` BitBake class[10] that searches a staged root filesystem for blacklisted files by name or by type. This class operates on a per-image basis by appending a `find_blacklisted_files` Python function to `${ROOTFS_POSTPROCESS_COMMAND}`.

The `${BLACKLISTED_FILE_NAMES}` variable is used to specify a semicolon-delimited list of filenames, optionally using glob syntax (e.g. `*.pem`), that should not be allowed in the root filesystem. For files that vary in their naming scheme, but have a known format, the `${BLACKLISTED_FILE_TYPES}` variable can instead be used. This semicolon-delimited

[10] Available online: https://github.com/nccgroup/yocto-whitepaper-examples

list should be assigned strings included in the output of the program "file" for any items that should be disallowed in the root filesystem.

The find_blacklisted_files() function iterates over the contents of a staged root filesystem, whose location is provided by the ${IMAGE_ROOTFS} variable [56]. For each encountered file, a case-insensitive comparison of its name is performed against the contents of ${BLACKLISTED_FILE_NAMES}. A warning is printed if a match is detected. Otherwise, "file -b" is invoked and the output is compared against the items specified in ${BLACKLISTED_FILE_TYPES}. Again, a warning is printed if a match is detected.

As an example, consider a situation in which private keys and source files have been deployed in firmware releases. Depending upon the nature of the data signed and/or encrypted with the private keys, access to the key material might allow an attacker to decrypt or tamper with sensitive information. Unintentional source code disclosure poses a risk to intellectual property. Ideally, it should not also pose a threat to system security, because credentials and key material should not be stored in source code in the first place. The following snippet from an ncc-demo-image_0.1.0.bb recipe contains ${BLACKLISTED_FILES_*} definitions that attempt to detect these items.

```
inherit rootfs-file-blacklist

# Blacklist private keys by known extension
BLACKLISTED_FILE_NAMES = "*.p12;*.pem;"

# Blacklist private keys and source files by `file` output
BLACKLISTED_FILE_TYPES = "private key;C source;"
```

**Listing 18:** File blacklist items added to an image recipe

Listing 19 shows the output generated by the rootfs-file-blacklist class during the build of ncc-demo-image. Observe that the disallowed items are reported as warning messages on the console. If these warnings are too easily overlooked, the bb.fatal() logging function could instead be used to force any detected items to forcibly break the build. A less drastic solution would be to both warn to the console, as well as log to a dedicated report file in ${DEPLOYDIR_IMAGE}. This would allow builds to complete, but still allow the presence of blacklisted items to be easily recorded as failures in automated build environments.

```
WARNING: ncc-demo-image-0.1.0-r0 do_rootfs: Blacklisted file (type) in rootfs:      ↩
    /usr/share/diag-server/default-key
WARNING: ncc-demo-image-0.1.0-r0 do_rootfs: Blacklisted file (type) in rootfs:      ↩
    /opt/fpga-utils/src/fpga_dumpregs.c
WARNING: ncc-demo-image-0.1.0-r0 do_rootfs: Blacklisted file (type) in rootfs:      ↩
    /opt/fpga-utils/include/fpga_regmap.h
WARNING: ncc-demo-image-0.1.0-r0 do_rootfs: Blacklisted file (name) in rootfs:      ↩
    /etc/mfg-util/certificate.p12
WARNING: ncc-demo-image-0.1.0-r0 do_rootfs: Blacklisted file (name) in rootfs:      ↩
    /etc/mfg-util/certificate.pem
```

**Listing 19:** Example rootfs-file-blacklist output

### 3.3.2 Per-Recipe Symbol Blacklist

Appendix I presents a symbol-blacklist class[11] that may be used to detect the use of undesired symbols (e.g. functions) within programs and libraries. This class could be highly beneficial in avoiding the reintroduction of problematic code if you have encountered situations such as the following:

- The use of "unsafe" variants of standard functions (e.g. strcpy, strcat, sprintf) resulting in systemic memory corruption defects

[11]Available online: https://github.com/nccgroup/yocto-whitepaper-examples

- The inclusion of privileged diagnostic functionality in release builds due to misconfigured build scripts or inconsistencies in preprocessor directive usages
- Extraneous logging functionality, introducing data disclosure risks

The `symbol-blacklist` class uses a `${BLACKLISTED_SYMBOLS}` variable to specify symbol names that shall be disallowed in all binaries produced by a recipe. The default definition of this variable contains functions that have traditionally been a source of security-impacting defects and have safer standard alternatives.

This class iterates over each item in a recipe's install destination directory (`${D}`), and invokes the appropriate cross-toolchain's `readelf` program on encountered ELF files to obtain a list of symbols. A warning is printed if a blacklisted symbol is detected in this symbol listing. As with the previous example, the manner in which blacklisted items are logged could easily be adjusted to suit your organization's needs.

Note that the `addtask` directive is used to automatically execute the `do_find_blacklisted_symbols` task after the installation task and before the packaging task. This task can also be invoked manually, via:

```
bitbake -c find_blacklisted_symbols <recipe>
```

**Listing 20:** Manual invocation of the `do_find_blacklisted_symbols` task from the command line

Listing 21 demonstrates how this BitBake class could be introduced into a recipe, additionally adding `log_data` and `log_verbose` to the symbol black list. Listing 22 presents example output, as obtained from the build of a "`symtest`" recipe.

```
inherit symbol-blacklist
BLACKLISTED_SYMBOLS += "log_data log_verbose"
```

**Listing 21:** Enabling the `symbol-blacklist` BitBake class in a recipe

```
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "log_data" present   ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/symtest/1.0-r0/image/usr/bin/ble-daemon
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "log_verbose" present ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi//1.0-r0/image/usr/bin/ble-daemon
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "strcpy" present     ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/symtest/1.0-r0/image/usr/bin/ble-daemon
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "stpcpy" present     ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/symtest/1.0-r0/image/usr/bin/ble-daemon
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "strcat" present     ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/symtest/1.0-r0/image/usr/bin/ble-daemon
WARNING: symtest-1.0-r0 do_find_blacklisted_symbols: Blacklisted symbol "sprintf" present    ↩
in ${TMPDIR}/work/arm1176jzfshf-vfp-poky-linux-gnueabi/symtest/1.0-r0/image/usr/bin/ble-daemon
```

**Listing 22:** Example `symbol-blacklist` output

Normally, when creating a class that scans a recipe's build artifacts, it makes the most sense to work with the recipe's build artifacts *after* they have been divided into their respective packages (e.g. `${PN}`, `${PN}-bin`, `${PN}-dev`). This allows for per-package control over the processing of build artifacts, which can be implemented by iterating over the post-package split directories located in `${PKGDEST}`. To do this, the associated task would need to be added after `do_package`.

However, this approach does not work in the situation where symbols in ELF files need to be inspected because the `do_package` task invokes a `split_and_strip_files` function that strips symbol names from binaries while dividing them up into their respective packages. Therefore, this example operates before the `do_package` task, just for simplicity. The astute reader might note that manipulation of `${PACKAGEFUNCS}` could instead be performed to insert custom QA checks before `split_and_strip_files` is called.

This `${D}` and `${PKGDEST}` caveat is highlighted here in order to stress the importance of determining an appropriate point in a recipe's build process to perform QA tasks, and to confirm that recipes and `bbappend` files are not altering desired behavior. For example, the build scripts included with some codebases may attempt to strip binaries after they are compiled. With a default configuration of `insane.bbclass`, Yocto will generate a QA error in this situation.

```
ERROR: QA Issue: File <file> from <recipe> was already stripped, this will prevent ↩
future debugging! [already-stripped]
```

**Listing 23:** Example `already-stripped` QA error message

The conventional solution to this issue is to create a patch for the software's build scripts to remove the post-build stripping of binaries, allowing the appropriate BitBake task to instead do this. However, one may encounter recipes that simply suppress this QA error, using the following line:

```
INSANE_SKIP_${PN}_append = "already-stripped"
```

**Listing 24:** A common, but unadvised, way to suppress an `already-skipped` QA error in a recipe

Such an implementation would yield false-negatives when attempting to use the `symbol-blacklist` class. This highlights the need to "sanity-check" custom QA tasks, which can typically be done using a few small test recipes.

# 4 System and Application Hardening

Attempting to detect and eliminate security defects is one important part of a successful product security strategy. This is an ongoing process over the lifetime of a product, typically consisting of architectural and implementation reviews, source code reviews, security testing, and the regular monitoring of security advisories associated with third-party components.

However, this process alone cannot guarantee that all potential security vulnerabilities will be found and resolved. Therefore, it is extremely important to mitigate remaining risks through additional "hardening" that aims to reduce the system's overall attack surface as well as limit the impact of any particular vulnerability.

This section highlights a few means to improve the overall security of a firmware image and apply fundamental security principles to further harden its components. Note that this **is not** intended to serve as a comprehensive Linux hardening checklist, but rather demonstrate how Yocto can be used to supplement or validate security hardening measures. Additional Linux hardening information can be found in [40,41,57–63].

## 4.1 Merging Backported Security Patches

The Yocto Project and the OpenEmbedded community regularly backport security patches to a supported subset of past releases.

Ensuring all of these patches are applied is a critical first step before beginning the process of hardening an image. Once firmware has been released, regularly monitoring the Yocto/OpenEmbedded release branches for security patches remains an important step in maintaining the security of a system.

Typically, the `git` changesets addressing CVEs contain a commit message explicitly noting the CVE identifiers associated with resolved security vulnerabilities. In situations where a recipe is updated to a new version of the upstream software, this usually consists of changing the recipe's filename (to adjust `${PN}`) and updating its `${SRC_URI}` `md5sum/sha256sum` definitions.

However, when security fixes are backported, they may instead be integrated into the build by adding one or more patches to a recipe's `${SRC_URI}` definition, referencing the included patches as local files (i.e. specified via "`file://`"). If the patch addresses a single CVE, then the filename typically includes the CVE identifier. Otherwise, if multiple vulnerabilities are resolved in a single patch file, that patch file should contain a list of fixed CVEs in the format:

```
CVE: CVE-<identifier> [CVE-<identifier> CVE-<identifier> ...]
```

**Listing 25:** CVE listing format commonly used in upstream Yocto and OpenEmbedded patch descriptions

These conventions are leveraged by the `cve-check` class (discussed in Section 3.1) to determine if a vulnerability has been patched.

## 4.2 Migrating Away from Reference Configurations

Per the Yocto Project documentation, the Poky reference distribution **is not** intended to be utilized directly in a product. Its purpose is only to serve as a "working example" used to bootstrap development [64]. Similarly, the `${MACHINE}` configuration files found in various layers tend to target reference design platforms that maximally demonstrate the various functionality offered by a SoC product line.

Therefore, an important part of transitioning from prototyping to product development is to migrate away from the reference configurations in favor of custom configurations tailored to your organization's needs. This allows for finer-grained control over settings and can aid in the removal of unneeded functionality that could contribute to the device's overall attack surface.

These tasks should be completed prior to investing significant time into hardening efforts. The custom configuration files will be one place where some changes will be made. Attempting to perform hardening tasks without having first pruned configuration files may also lead to wasted effort.

### 4.2.1 Distribution Configuration

By default, the Yocto releases use the Poky reference distribution defined in the `meta-poky/conf/distro/poky.conf` file. As outlined in the Yocto documentation [65], creating a custom distribution consists of adding a configuration file to a `conf/distro` directory within your custom layer. The name of this configuration file represents the distribution name; this must be specified (without the `.conf` suffix) via the `DISTRO` variable in your `local.conf` file in order to select your custom distribution.

It is highly recommended that the default settings in your custom distribution configuration represent the production-ready state. This helps prevent unintentional variations in successive firmware releases and limits the required changes in `local.conf` to the definitions of `${DISTRO}` and `${MACHINE}`.

Instead of having development team members enable diagnostic functionality through ad-hoc variable definitions in their `local.conf`, consider maintaining one or more variants of your custom distribution that enable different degrees of debug capabilities. By placing a majority of the custom distribution configuration within include (`conf/distro/include/*.inc`) files, the same core configuration can be shared amongst these variants via `require` statements; the bulk of each resulting distro configuration file represents the variant-specific changes used to enable the desired diagnostic functionality.

Refer to the files included in the `poky/meta/conf/distro` and `poky/meta-poky/conf/distro` directories to gain an understanding of the configuration options that make up the Poky reference distribution. For an additional example, see the configuration file for the Angstrom distribution [66] that is built using OpenEmbedded. While the items that need to be added, removed, or changed will vary from one organization to another, the following variables may be of interest. See the Yocto Project Reference Manual's *Variables Glossary* [67] for more information about each of these.

- `${DISTRO_FEATURES}`, `${IMAGE_FEATURES}` - Unnecessary or extraneous features can increase the overall attack surface of a system. Include only the minimum set of features required to meet the baseline requirements for the particular product or family of products that your custom distribution will be supporting.
- `${DISTRO_EXTRA_RDEPENDS}`, `${DISTRO_EXTRA_RRECOMMENDS}` - For the reasons noted above, only the minimum required set of recipes should be included via these variable definitions.
- `${WARN_QA}`, `${ERROR_QA}` - The Poky distro configuration demonstrates how to enforce stricter QA by assigning a `${WARN_TO_ERROR_QA}` variable with a list of QA tasks to elevate from warning to errors. It then removes these from `${WARN_QA}` via `WARN_QA_remove` and adds them to `ERROR_QA` via `ERROR_QA_append`. You can expand upon these variable definitions to further tune the QA issue severities and add your own organization-specific QA tasks.
- `${PREFERRED_PROVIDER}`, `${PREFERRED_VERSION}` - In cases where multiple recipes provide a particular package, or different versions of the same package, these definitions may be used to indicate the desired recipe. When using the wildcard character (%) in a `${PREFERRED_VERSION}` assignment, ensure that the allowed version number variability will not unexpectedly introduce new functionality. Although many software projects use a patch version number to denote backwards-compatible bug and security fixes that do not introduce new functionality, not all projects use a semantic version numbering scheme. Avoid using wildcards on version control changesets (e.g. `+git%`) or the `AUTOREV` feature; these inherently conflict with the goal of creating well-defined and reproducible production release builds.
- `${PNBLACKLIST}` - If there are specific recipes that should never be included in in an image, this variable may be used to produce a build error (with a configurable message) when attempting to build a blacklisted recipe. This feature can help ensure that production release images are not contaminated with internal development or diagnostic tools.
- `${MIRRORS}` and `${PREMIRRORS}` - When using mirrors, consider any risks associated with internal recipe names being queried on the specified mirrors (i.e. potentially leaking future product or feature names in third-party mirror logs). If this is a concern for your organization, `own-mirrors.bbclass` [68] can be used to point to your organizations own internal source mirrors. Alternatively, a custom class could be used to clear recipes' `${PREMIRRORS}` variable, preventing mirrors from being used.

#### 4.2.2 Reference Machine Configurations

Reference machine configuration files, found in layers' `conf/machine` directory, also may contain features and dependencies that are extraneous in the context of your product requirements. As noted above, it is important to remove unnecessary items before investing time in hardening efforts. When migrating to a custom machine configuration file, be sure to review both the reference machine configuration file you were using directly, along with the files it included via `require` or `include` statements. As a matter of best practice, definitions common to a particular SoC or family of SoCs are typically consolidated within one or more include files.

Machine-related variables of interest include the following. See the BSP Machine Configuration Example [69] for sample usages of these and additional variables to review.

- `${MACHINE_FEATURES}` - Similar to the distribution features variable, this should be reduced to the minimum required set of features.
- `${MACHINE_EXTRA_RDEPENDS}`, `${MACHINE_EXTRA_RRECOMMENDS}` - Again, the recipes specified by these definitions should be kept to a bare minimum.
- `${SERIAL_CONSOLE}`,[12] `${SERIAL_CONSOLES}` - These definitions are used to configure the use of one or more serial consoles as TTYs (via getty). On a production release configuration, set `${SERIAL_CONSOLES}` to an empty string to disable logins from the console or logging to a TTY. This largely amounts to a matter of best practice. Production firmware releases should not have users configured with the ability to log in, nor should sensitive information ever be logged. Review the following recipes to better understand how these variables are used:
  - `busybox-inittab`
  - `shadow-securetty`
  - `systemd-serialgetty`
  - `sysvinit-inittab`
- `${USE_VT}` - This variable is used in the `sysv-inittab` recipe to control whether or not getty is run on any virtual terminals. If set to `"1"`, getty inittab entries are created for the TTY identifiers specified in `${SYSVINIT_ENABLED_GETTYS}`. In production releases, set `USE_VT = "0"` and `SYSVINIT_ENABLED_GETTYS = ""`.
- `${UBOOT_MACHINE}`, `${KMACHINE}` - These variables map the machine name used in the configuration file to those used by the U-Boot and Linux kernel build systems, respectively. These should point to your organization's custom targets, rather than a reference design platform.
- `${KERNEL_DEVICETREE}` - This variable should refer to your platform's device tree BLOB (`.dtb` file) built along with the Linux Kernel. The associated `.dts` file should be customized for your platform(s), only including and enabling peripherals and interfaces on an as-needed basis.

### 4.3 Enabling Exploit Mitigations in All Recipes

As discussed in Appendix B, modern compilers provide a number of exploit mitigation features that can make security vulnerabilities significantly more difficult to exploit reliably or on a large scale. OpenEmbedded and Yocto provide a `security_flags.inc` file, which can be used to enable these compiler features across all recipes in a firmware image by updating the definitions of the `${CFLAGS}` and `${LDFLAGS}` variables used during compilation tasks.

This can be included in a custom distribution configuration (or `local.conf`) file as follows:

```
require conf/distro/include/security_flags.inc
```

**Listing 26:** Enabling the use of security_flags.inc

[12] `${SERIAL_CONSOLE}` is now deprecated. `${SERIAL_CONSOLES}` should be used instead.

This file defines SECURITY_CFLAGS and SECURITY_LDFLAGS variables, which are appended to the TARGET_CC_ARCH and TARGET_LD_ARCH variable definitions used during cross-compilation for the target architecture. The following snippet presents the default SECURITY_* flag definitions found in security_flags.inc.

```
GCCPIE ?= "--enable-default-pie"
# Inject pie flags into compiler flags if not configured with gcc itself
# especially useful with external toolchains
SECURITY_PIE_CFLAGS ?= "${@'' if '${GCCPIE}' else '-pie -fPIE'}"

# _FORTIFY_SOURCE requires -O1 or higher, so disable in debug builds as they use
# -O0 which then results in a compiler warning.
lcl_maybe_fortify = "${@oe.utils.conditional('DEBUG_BUILD','1','','-D_FORTIFY_SOURCE=2',d)}"

# Error on use of format strings that represent possible security problems
SECURITY_STRINGFORMAT ?= "-Wformat -Wformat-security -Werror=format-security"

SECURITY_CFLAGS ?= \
"-fstack-protector-strong ${SECURITY_PIE_CFLAGS} ${lcl_maybe_fortify} ${SECURITY_STRINGFORMAT}"

SECURITY_LDFLAGS ?= "-fstack-protector-strong -Wl,-z,relro,-z,now"
```

**Listing 27:** Security flag variable definitions in security_flags.inc

Some software will fail to build with all of the above flags. This can be addressed in a few ways:

- Refer to the upstream source repository to determine if the build failures have been addressed in later releases. Either update to that version, or backport a patch to be included by the associated recipe.
- Review the associated packaging scripts or source packages from popular Linux distributions. These may already contain patches to enable the problematic flags.
- Disable the use of the problematic compiler flag(s) for that specific software.

Although the first two options are preferable from a security perspective, they may not always be possible.

When the last option is used, especially in the early phases of a project, consider opening an item in your organization's issue tracker to log this workaround; it is very easy to forget to revisit these and lose track of the degree to which such workarounds are utilized. As recipes are updated over time, these workarounds should be periodically revisited to determine if they are still necessary and removed when no longer required.

The full security_flags.inc file contains a few variants of these variable definitions, with different options removed. Per-recipe definitions of the SECURITY_* items can instead be provided, using the appropriate subset of compiler flags.

For example, the following snippet from security_flags.inc demonstrates how valgrind is compiled without Position Independent Executable (PIE) flags, and how busybox is compiled without treating format string warnings as errors.

```
SECURITY_CFLAGS_pn-valgrind = "${SECURITY_NOPIE_CFLAGS}"
SECURITY_STRINGFORMAT_pn-busybox = ""
```

**Listing 28:** Per-package security flag variable overrides

Lastly, build scripts for a particular piece of software may not always respect the CFLAGS and LDFLAGS definitions provided by the BitBake environment. Such software is often conspicuous due to build failures when targeting non-x86 architectures. Otherwise, the checksec script discussed in Section 3.2 can help identify these items. Resolving this issue typically consists of creating a patch for the problematic files and including this patch in a recipe's SRC_URI.

## 4.4 Implementing Privilege Separation

In general, the security of an entire system should not be undermined by the compromise of any single component. This is similar to fault-tolerant design principles that aim to sustain system operations despite random faults in one or more components. One key difference in the security context, however, is that one must assume that attackers will actively attempt to induce and exploit specific failure modes in order to achieve one or more desired behaviors.

A common approach in the design of robust systems is to decompose an architecture into smaller, maximally cohesive components, with each one minimizing reliance upon others when performing its dedicated function. Combining this approach with the security principle of *least privilege*[13] [70] can yield effective privilege separation within a software architecture. This is often referred to as "compartmentalization." From the security perspective, the interfaces between architectural components represent trust boundaries; any information exchanged between components must be regarded as "untrusted" and be validated for authenticity and correctness prior to use.

Examples of systems lacking privilege separation can often be found in consumer-grade networking equipment, which often run web management servers as a privileged user (e.g. root). Although this may greatly simplify application code when attempting to interact with drivers and associated hardware peripherals, this introduces a dangerous single point of failure in the overall security architecture. As a result, the successful exploitation of common vulnerability classes, such as Command Injection [71] and Cross-Site Request Forgery (CSRF) [72], quickly lead to the complete compromise of the system. A real-world example of this is presented in [73].

### 4.4.1 An Example Device Configuration Architecture

A more security-oriented design that exemplifies privilege separation is shown in Figure 2. In this architecture for an imaginary Internet of Things (IoT) device, both a web server and a wireless protocol daemon require the ability to read and write device configuration values stored in an EEPROM accessed via `/dev/i2c-0`.[14] However, they are not permitted to directly access this storage device. Instead, both the web-server and wireless daemon processes execute under their own unique and limited-privilege UIDs. These applications request access to configuration data through the Configuration Manager daemon, which has sufficient privileges to read and write the `/dev/i2c-0` device node. By leveraging the `SO_PEERCRED` Unix Domain socket feature [74], the Configuration Manager is able to obtain the UIDs of the processes it is communicating with .[15]

This allows the Configuration Manager to grant or deny requests to read or write configuration data on a per-UID basis. This not only provides a means to enforce access controls over which configuration data a requesting process can access, but also reduces the impact of a compromise of the Web Server and Wireless Protocol Daemon components.

Of course, significant risk still remains if the Configuration Manager is permitted to execute as an overly-privileged user. For example, a message parsing defect might yield an exploitable memory corruption vulnerability that an attacker can leverage to execute arbitrary code with the privileges of this process. This risk can be mitigated in a few ways.

First, the Configuration Manager itself could be executed under its own dedicated UID, and the `/dev/i2c-0` device node could have its file ownership and permissions adjusted accordingly. If multiple software components require access to this device node, then a GID-based restriction may be more appropriate.

In situations where access to a resource cannot be restricted by file permissions alone, it may be necessary to launch a process as a privileged user, acquire handles to the necessary resources, and then relinquish the privileges that are no longer needed by dropping to a less-privileged UID/GID [39]. This is often easier to do when the process is being

[13]The principle of least privilege requires that a component in a system shall have only the minimum level of access to information and resources required to fulfill its requirements. Adhering to this principle helps minimize the impact of a single component becoming compromised.
[14]For simplicity, assume that this is the only device on this I2C bus. If multiple devices were present, additional access controls might be needed to restrict applications to their respective devices.
[15]This approach is effective specifically because the low-privileged user space processes sending their credentials in Unix Domain Socket metadata cannot "spoof" this information - it is validated by the kernel. A process executing as root, on the other hand, is permitted to specify any identifiers.

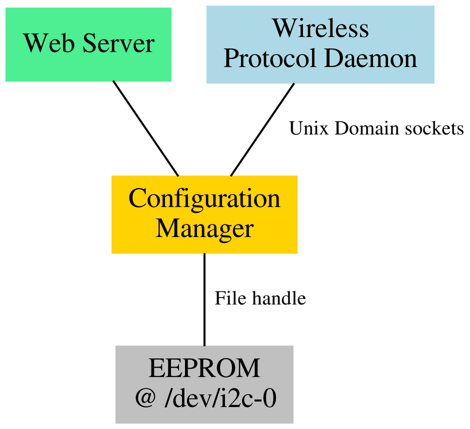

**Figure 2:** A simple architecture for enforcing per-UID access controls on data stored in an EEPROM

launched as a part of normal system startup procedures. However, if the process is to be launched at later time, this may result in the undesirable introduction of setuid/setgid binaries into the system.

As noted earlier in Section 3.2, Linux capabilities [42] can instead be used to reduce the abilities of a privileged process. Seccomp [43] can also be used to limit the system calls that a process is permitted to use. Of course, for these technologies to be maximally effective, one must have first performed the design decomposition exercise noted at the beginning of this section. These technologies can be difficult to integrate effectively when dealing with massive, monolithic software components that implement a broad range of functionality.

### 4.4.2 User and Group Allocation in Yocto

The `useradd` family of BitBake classes [75] can be used to allocate users and groups on a per-recipe basis, allowing for the privilege separation described in the previous sections to be achieved. As noted in [75], the `useradd-example.bb` recipe exemplifies this use of this class. Other recipes using the `useradd` class include the following:

- `meta/recipes-core/dbus/dbus_1.12.2.bb` - The recipe allocates a `messagebus` user and group, along with a `netdev` group.
- `meta/recipes-core/systemd/systemd_237.bb` - The `systemd` recipe conditionally adds per-service users based upon deployed packages. This also demonstrates adding separate users on a per-package basis (i.e. for `${PN}` and `${PN}-extra-utils`).
- `meta/recipes-connectivity/dhcp/dhcp.inc` - This include file for the `dhcp` recipe creates a dedicated `dhcp` user for the DHCP server provided by the `dhcp-server` package.
- `meta/recipes-graphics/wayland/weston_3.0.0.bb` - This recipe creates a `weston-launch` system group.
- `meta/recipes-extended/rpcbind/rpcbind_0.2.4.bb` - The `rpcbind` recipe creates an `rpc` user and group, and specifies this group name during the configuration process via the `--with-rpcuser` item in its `EXTRA_OECONF` definition.

By default, the build system assigns UIDs and GIDs dynamically. This implies that applications, such as the Configuration Manager in the previous example, cannot hard-code these identifiers. Instead, programs running on the target would need to use standard functions, such as `getpwnam_r()` [76] to determine the ID values associated with user and group names.

This dynamic behavior may be undesired, especially if it results in ID changes as new features and services are introduced in later firmware releases. If this is the case, the `${USERADDEXTENSION}` feature [77] may be used to supply `passwd` and `groups` files containing statically-defined IDs. Although the documentation states that this variable may be set in one's `local.conf` file, consider instead using your custom distribution configuration file (see Section 4.2.1).

The `extrausers` BitBake class [78] allows user and group allocations to be performed image-wide, as opposed to a per-recipe basis. Heavy reliance on this class may indicate poor modularization of recipes, or architectural decisions requiring further review and scrutiny. It is very often the case that per-recipe user/group allocations and properly specified dependencies can be used to achieve the same results.

When allocating users for various applications and services, do not configure password-based login for these users. There is rarely a use-case that requires interactive console logins for these types of users. Note that there is an extremely important difference between disabling passwords (via a `*` or `!` in `/etc/shadow`) and empty passwords. The former, which is the default behavior when `useradd` is invoked without a `-p/--password` argument, prevents successful logins because no hash will match these values. An empty password entry implies that no password is required to authenticate, which is certainly undesirable in production firmware releases.

It is a security best-practice to set these users' shell to a program that performs no functionality if a login were to succeed. The `/usr/sbin/nologin` and `/bin/false` programs are commonly used.

## 4.5 Reducing Devices' Attack Surface

Attack surface reduction is an important part of the system security hardening process. The goal of this effort is to eliminate extraneous attack vectors and potentially viable exploitation methods, including those that become available only after a successful compromise of other portions of a system. Reducing an embedded platform's attack surface largely consists of disabling unused features at build-time and limiting the availability of exposed interfaces at run-time.

Although the removal of unused functionality can incur a non-trivial time investment, it is one that can provide significant benefit beyond improving security. The elimination of unnecessary "moving parts" can significantly reduce the size of firmware images, as well as reduce memory and CPU footprints. Treat the removal of unneeded functionality as an opportunity to reduce technical debt and mitigate risks associated with less-reviewed portions of a system and any unknowns.

Bear in mind that reducing a device's attack surface does not *only* consist of build-time decisions. Runtime decisions about when to enable and disable a communications subsystem, or how long to continue accepting input on a particular interface, can also reduce a system's overall attack surface. The build-time changes are simply more relevant to this paper, and therefore are the focus of the suggestions presented here.

This section first covers how configuration changes can be applied, and then outlines a few general types of changes that can reduce a platform's attack surface. It is by no means comprehensive, as the specific requirements of a product will largely dictate what is necessary. In general, any features or interfaces that aid in development, debugging, and diagnostics can greatly aid an attacker; these should all be treated as potential threat vectors to remove.

If diagnostic functionality cannot be removed, then it should be protected by strong authentication that is tightly coupled to a specific unit; the complete compromise of one platform should not imply that an attacker can immediately access privileged functionality on another platform. Furthermore, end users' data should never be placed at risk when entering a diagnostic mode. A system should always the enforce secure erasure of user data prior to exposing privileged functionality to service technicians or factory personnel.

Examples of insecure, but common, protections include the use of product-wide hard-coded passwords and credentials trivially derived from device identifiers (e.g. serial number, MAC address). Also consider the impact of any single device's private key(s) becoming compromised when designing such a mechanism. A better solution might require a service technician to first authenticate to a device using a device-unique key pair and a challenge-response mechanism before transitioning to a diagnostic state. Implementing factory mode "unlock" functionality in a manner that creates an audit trail and requires connectivity to a trusted system can help detect and mitigate abuse of this functionality, as well as restrict its use to a designated repair facility. For information about protecting devices at various stages in the production and support life cycle, refer to [79].

### 4.5.1 Customizing Build and Runtime Configurations

The default build configurations in recipes, as well as runtime configurations deployed by them, will likely include extra functionality than is necessitated by product requirements. Modifications to these configurations can generally be made through the use of `bbappend` [25] files. This is highly recommended over modifications to third-party layers and recipes; isolating these changes to `bbappend` files within your organization's own version-controlled layers can help improve traceability and avoid issues stemming from different teams not having the same patches applied to third-party layers.

The ease of making changes to compile-time configurations largely depends upon the build system being used by the target software. If autotools or CMake are used, flags to disable features can typically be appended to the definitions of `${EXTRA_OECONF}` or `${EXTRA_OECMAKE}`, respectively. See [80] for a list variables related to defining "extra" build information in recipes.

Although it is possible to remove items from these variable definitions using BitBake's "`_remove`" suffix [81], care must be taken when doing so. It may not always be immediately clear how a variable definition is being constructed or modified by various levels of include files or even `bbappend` files from other layers. If significant changes to configuration values are required, it is often better to entirely override and redefine the variable in your `bbappend`, ensuring that it contains exactly what is desired.

When only a few changes to a default configuration file are required, it may be tempting to implement a small `do_configure_append()` function that uses `sed` to toggle the state of relevant settings; this is common in upstream OpenEmbedded and Yocto layers. However, this yields a potential risk of upstream recipe updates changing the configuration in a manner that results in an increased attack surface. Your organization may find that maintaining their own complete configuration files within their own layers is a more reliable way to ensure that software configurations do not change silently and unexpectedly. See [82] for an example of how one can completely replace the `/etc/network/interfaces` file deployed by the `init-ifupdown` recipe.

As upstream software and corresponding recipes are updated over time, your build configuration changes and customized files might result in build or functional test failures. This is inherently a trade-off; from the security perspective, a build or test failure prompting a configuration review is safer than allowing new functionality to be deployed without adequate review.

### 4.5.2 Image and Machine Features

The various `FEATURES` variables described in [38] control how various recipes are configured and deployed in a target system. Especially when migrating from a reference platform and the Poky reference distributions, these are important to review in order to remove extraneous functionality.

In particular, ensure that `EXTRA_IMAGE_FEATURES` items used during development are not used in production firmware releases. A number of these settings, such as `allow-empty-password`, `debug-tweaks`, `empty-root-password`, and `tools-debug` trade off security for a development-friendly environment.

### 4.5.3 Network Interfaces and Firewall Configuration

As previously noted, the example in [82] presents a straight-forward approach for deploying a custom `/etc/network/interfaces` file. The default `interfaces` file configures multiple network interfaces to obtain IP addresses via DHCP. However, only `lo` and `eth0` are configured to be automatically initialized (via `auto`).

For a product without networking functionality, wireless interfaces, or Ethernet ports, these extraneous configuration items may seem harmless. However, the potential risks become clear when the same system provides USB ports to end users and internally uses network sockets (bound to `0.0.0.0`) as IPC mechanisms. By attaching a USB Ethernet adapter, an attacker may be able to gain access to internal IPC interfaces. If the applications using these socket-based IPC interfaces do not properly validate received data, this unintentional exposure could quickly yield security vulnerabilities. Although this example may seem unlikely, it is inspired by real-world scenarios and demonstrates the value of disabling unnecessary network interfaces.

Even when a product is intended to be deployed on customers' internal networks, it should be assumed that devices will be deployed on hostile networks. As shown in publicly available scans [83], many devices are (unintentionally) exposed to the Internet. Thus, it is prudent to deploy systems with restrictive firewall settings, and ensure that server applications listen only on the intended interfaces. Yocto Poky releases provide an `iptables` recipe in `meta/recipes-extended` [84]. Iptables rules can be loaded via an `iptables-restore` invocation included in an initscript or as a `pre-up` argument in the `/etc/network/interfaces` file. Be sure to include rules for both IPv4 and IPv6, and include tests of both in your QA procedures.

### 4.5.4 Consoles and Interactive Logins

The exposure of privileged consoles over UART interfaces is still incredibly common in consumer, enterprise, and industrial-grade products. Although they may be used heavily during development, they are very rarely necessary in a final product. These interfaces present a security risk when attackers may have physical access to a device. They also provide an avenue for reverse engineering and vulnerability discovery to persons with hobbyist-level capabilities. One common argument for these types of interfaces is the need for factory personnel and engineers to diagnose failures on returned devices. If root-level access is a requirement for RMA processes, then it is almost always a better approach to require strong authentication and the secure erasure of user data before allowing a device to enter a less secure state. From this state, the authorized personnel could then load an authenticated factory test image to perform diagnostics.

As noted in Section 4.2.2, changes to variable definitions of `${SERIAL_CONSOLE}`, `${SERIAL_CONSOLES}`, `${USE_VT}`, and `${SYSVINIT_ENABLED_GETTYS}` are used to enable a console over a serial port. Clearing these definitions should remove the associated getty configuration items. This, combined with the removal of the items noted in Section 4.5.2 is generally sufficient to remove local interactive login functionality . Of course, recipes change over time and this should be confirmed through testing, as well as review of the `/etc/shadow`, `/etc/passwd` and `/etc/securetty` files.[16]

The same principles also apply to authenticated network-based login mechanisms, such as the `dropbear` ssh server, which is commonly enabled in development builds. For many products, an ssh server is superfluous and should be removed from production releases. If required for a product, privileged users should not be allowed to log in through this interface, and only strong key-based authentication should be allowed. Again, consider the impact of private key compromises on an entire product line when making use of such technology, and try to minimize this impact.

### 4.5.5 BusyBox

BusyBox implements a variety of common Unix utilities within a single size-optimized binary [85]. Historically, this was deployed as a set-UID (SUID) root-owned binary, which inherently carries the risk of privilege escalation if an exploitable defect is identified. As of the Yocto 1.5 Dora release (circa 2014), the BusyBox recipe sets a default `${BUSYBOX_SPLIT_SUID}` value of `"1"`, which produces separate setuid and non-setuid versions of BusyBox, with the former containing only a minimal number utilities. Avoid disabling this split-SUID functionality, and treat changes to `${BUSYBOX_SPLIT_SUID}` as potential red flags.

The default BusyBox `defconfig` file provided with Yocto Poky releases enables a number of utilities that are helpful during development, but are often not required in a final product. Some of these items may prove particularly useful to an attacker, should they identify a means to execute arbitrary commands on a system [71,73]. The removal of extraneous commands, such as editors, networking utilities, and development tools may help limit impact of an otherwise effective attack, or at least raise the bar for successful exploitation. The BusyBox `defconfig` can be replaced via a `bbappend` file, as done for an `interfaces` file in Listing 8. Some items to consider for removal in release builds include `dc`, `dd`, `microcom`, `nc`, `telnet`, `tftp`, `wget`, and `vi`. Of course, the complete list will depend upon your product's requirements and there may be many more utilities that can be safely removed.

### 4.5.6 Linux Kernel

The Linux kernel configurations accompanying reference design platforms are usually configured to support development, and are not indicative of a hardened configuration. Although a detailed discussion of Linux kernel hardening

[16]This is a good candidate for a custom QA task (see Section 3.3).

is outside the scope of this paper, removal of development and debug functionality, along with disabling extraneous subsystems, are good first steps. Even in cases where user-space configurations are already effective in disabling undesired functionality, supplementing this with the kernel-level removal of functionality can provide another level of assurance that later changes, potentially by another team, will not (re)introduce undesired behavior.

Work on a kernel configuration is best done iteratively using functional tests because it is not uncommon to inadvertently disable required functionality during this process. As described in Section 4.2.2, it is advisable to create a custom machine configuration file, with a customized kernel configuration that is specified by the `${KMACHINE}` variable. The Kernel Configuration Analyzer described in Section 3.2.2 may be used to help identify potentially dangerous items. Below are just a few items to strongly consider disabling:

- DebugFS provides access to kernel information from user space, and is intended for use by kernel developers.
- The `/dev/mem` and `/dev/kmem` devices provide direct access to physical and kernel memory, respectively. The former is sometimes used to map an SoC peripheral's register space into a user space process.
- Extraneous filesystem implementations, such as networked filesystems, are often enabled by default.
- The KProbes debugging feature supports event monitoring and the dynamic insertion of instrumentation callbacks in kernel code. Another debugging feature called "Ftrace" allows function calls within the kernel to be traced.
- Subsystem and driver-specific debug functionality may expose interfaces and information not intended for production systems. These are often controlled by `CONFIG_*_DEBUG` KConfig values.
- The Magic SysRQ feature allows a particular keyboard combination (Alt+SysRq) to be used to trigger low-level actions within the kernel, such as changing the kernel console log level, forcing a crash dump, displaying task information, and outputting a backtrace.
- `CONFIG_BUG` enables backtraces and register dumps for `BUG()` and `WARN()` calls. `CONFIG_KALLSYMS` includes symbolic information in backtraces. Should an attacker identify opportunities to trigger backtraces, this information can prove useful in allowing them to better understand potential vulnerabilities.
- `CONFIG_IKCONFIG` and `CONFIG_IKCONFIG_PROC` enable in-memory storage of the kernel configuration and expose it through a `/proc/config.gz` procfs entry. This unnecessarily exposes information about the kernel configuration.

Bear in mind that the Kernel Configuration Analyzer and the brief list presented above are not comprehensive, and do not take your platform's specific features and requirements into account. When evaluating your kernel configuration and reviewing recommendations from various sources, first focus on the motivations of the recommendations you encounter, rather than simply aggregating a list of KConfig values to toggle.

Features that allow an attacker to gain additional insight into the state of the kernel from user space should be regarded as candidates for removal. Functionality that could allow an attacker to manipulate or alter the state of the kernel in unintended ways should certainly be disabled. Debugging, instrumentation, and profiling features tend to be examples of both. The Linux kernel codebase is extremely large, and a thorough review of code used by your platform is unlikely to be practical. The removal of unused functionality is an opportunity to mitigate any potential unknowns and risks associated with that functionality.

Finally, kernel space versions of some of the user space exploit mitigations detailed in Appendix B, such as stack protection and ASLR, are present in modern kernel versions and should be enabled. Commercially available security enhancement patches are also available [86] and can be utilized to help mitigate remaining risks. For more information about Linux kernel hardening, see [63,87–89].

#### 4.5.7 U-Boot

A similar approach of removing extraneous compile-time configuration items can also aid in reducing the U-Boot bootloader's attack surface. Again, migrating away from reference design configurations and creating a custom machine (referred to by `${UBOOT_MACHINE}`) and associated U-Boot `defconfig` is an important step in this process.

Two common U-Boot attack vectors are its UART command shell and environment variables stored in flash. U-Boot's hush shell is extremely helpful during board bring-up tasks; it provides a variety of commands for interacting with and testing peripherals, loading images of various formats, and directly manipulating the contents of memory. However, when left in production releases, this functionality can provide an attacker with a means to extract and tamper with data, as well as execute malicious code.

Some devices have attempted to disallow access to the U-Boot console by setting the compile-time `CONFIG_BOOTDELAY` value to `0`. Prior to U-Boot 2016.07, the state of the `CONFIG_ZERO_BOOTDELAY_CHECK` value determined whether or not a keystroke could still be used to abort the autoboot sequence and drop to the interactive hush shell console when the boot delay was set to `0` [90]. With the 2016.07 release, the `CONFIG_ZERO_BOOTDELAY_CHECK` preprocessor macro conditionals were removed. Autoboot with no delay and no key-press interruption is now only implemented by `CONFIG_BOOTDELAY=-2` and a value of `0` always allows autoboot to be interrupted. As a result, if an out-of-tree platform's `CONFIG_BOOTDELAY` was not changed from `0` to `-2` when updating to a new version of U-Boot, autoboot interruption would become unintentionally enabled.

Even with autoboot interruption disabled, the U-Boot console may still be accessible if the sequence of commands used to boot the system[17] "fail open" as a result of an unexpected failure, exposing the interactive console. The published "rooting" process for a number of devices involves inducing this failure mode by temporarily shorting an I/O pin of an external storage device to ground while its contents are being accessed [91–93]. After this failure mode is successfully triggered, modified kernel arguments or filesystem contents can be used to obtain access to the operating system's root user account.

To eliminate this attack vector, disable the interactive console (see `CONFIG_CMDLINE`), and implement the boot processes programmatically within `board_run_command()` [94]. Of course, this implementation must not utilize untrusted data from external sources. An example of a `board_run_command()` implementation may be found in the board initialization code for the "USB armory" platform [95]. Also consider silencing the U-Boot console output via the `SILENT_CONSOLE` option or by removing UART support entirely to avoid disclosing information about the boot process.

U-Boot environment variables can be used to override compile-time defaults, and may be saved to a dedicated region on an external storage device. Implicitly trusting these environment variables presents a security risk when offline modification is possible. For example, if the `bootargs` variable passed to the Linux kernel are populated from variables stored in an external flash, an attacker might remove the flash part, modify the environment variables to insert "`init=/bin/sh`" in the `bootargs` definition, and then repopulate the part. If successful, this would boot the system directly to a root console.

To prevent this attack, the reliance upon on the untrusted environment could be entirely eliminated. The type of device that stores the U-Boot environment is defined by compile-time settings whose names begin with "`CONFIG_ENV_IS_IN_`." A `CONFIG_ENV_IS_NOWHERE` option is also available, which provides a dummy configuration that forces the use of the default, compile-time-defined environment [96,97]. Alternatively, a means to authenticate the externally stored environment could be implemented. However, the design and implementation of such security-critical functionality is well beyond the scope of this paper; it is a non-trivial task that presents numerous potential pitfalls.

## 4.6 Additional Topics

So far, this section has covered patching, insecure development configurations, exploit mitigations, privileged separation fundamentals, and attack surface reduction. However, there are plenty of additional security topics and technologies that warrant further exploration. The remainder of this section briefly highlights two additional topics that your organization should further investigate - Mandatory Access Controls and Secure Boot. Neither of these technologies can simply be "bolted onto" existing products, however. Their implementations require important design decisions to be made very early on in the product life cycle. Due to their complexity, these and other topics, such as trusted execution environments (TEEs), cannot be sufficiently covered here. Therefore, the remainder of this section aims only to introduce the two aforementioned concepts and to provide references to additional information.

[17] This is typically defined by the compile-time `CONFIG_BOOTCOMMAND` string and the runtime `bootcmd` environment variable.

### 4.6.1 Mandatory Access Controls

Traditional Unix file-based permissions are a form of Discretionary Access Control (DAC), in which security is enforced by the owner of a resource. In this model, an owner is able to make security policy decisions regarding that object, such as changing it to have more or less restrictive permissions. In contrast, a system employing Mandatory Access Controls (MACs) has security policies controlled by a policy administrator; users do not have the ability to override policy decisions, even when they are a superuser (e.g. root). MAC implementations tend to facilitate finer-grained access controls than DAC implementations. MAC policies can be used to restrict the types of actions that a particular process or user can perform on a resource, whether it be a file, directory, device, or socket.

Two popular MAC implementations for Linux are AppArmor [98–100], and SELinux [101–103]. A modified version of the latter has been used to enforce policies in the Android security model since version 4.4 (KitKat), released in late 2013 [104]. An AppArmor recipe is available in the `meta-security` layer [105] and SELinux support is available in the `meta-selinux` layer [106]. These layers provide reference policies that can be used to confirm basic functionality. However, the security policies used for your final product will need to be highly tailored to your system's architecture and security requirements. Attempting to implement and deploy MAC policies before performing a security-focused architectural review will very likely result in inconsistent policy implementations and wasted effort. Therefore, it is critical to define roles and access controls early in the design process.

### 4.6.2 Secure Boot

To help protect against malicious firmware modifications, many SoC vendors provide a "secure boot" mechanism in their products. The goal of these mechanisms is to ensure that only trusted, unmodified code is executed by the processor. Typically, this is achieved by having a ROM-based bootloader read the next boot stage into internal RAM and then verify the loaded code's cryptographic signature before executing it. The public key(s) used in the signature verification, version metadata used to prevent roll-backs, and the actions taken by the boot ROM in the event of a verification failure are often defined by the values stored in internal one-time programmable (OTP) memory.[18] Failure to set OTP secure-boot configuration values correctly in production systems could allow a device to fall back to insecure boot modes on alternative interfaces, allowing unauthorized code to be executed.

Once a boot ROM has handed off execution to another next stage of the process, it is the responsibility of each successive stage to verify the authenticity of any code and data that is loaded.

Below is a very simplified overview of what this process might look like. Version rollback checks have been omitted for brevity.

1. The processor is powered on and execution begins in a trusted program stored in internal ROM.
2. The boot ROM program checks the state of OTP configuration values to determine which external interface is used to load U-Boot. It configures the required peripheral interface and uses it to load a signed U-Boot image into internal RAM.
3. The boot ROM code loads a public key stored in OTP memory and uses it to verify that the U-Boot image is cryptographically signed by the corresponding private key. If the verification fails, the device enters an error handler function and performs no further action.
4. If verification succeeds, U-Boot begins executing from internal RAM and configures the processor's external RAM interface, along with any additional peripherals required to boot Linux.
5. U-Boot loads the Linux kernel and any required metadata[19] from external storage into RAM. The authenticity of the kernel and the accompanying metadata are established via their cryptographic signatures. Again, if any verification steps fail, the system enters an error state and performs no further actions.
6. The Linux kernel is booted from RAM, using the authenticated metadata to establish the integrity and authenticity of the root filesystem.

[18]Some vendors refer to flags in these regions as "fuse bits."
[19]Examples include device tree BLOBs, kernel command-line arguments, and dm-verity hash tree metadata.

At the end of this process, code executing from the root filesystem has the assurance that system contents have not been tampered with. If other sources of unverified input are used during this process, such as commands from the U-Boot shell or additional configuration data stored elsewhere in flash, it quickly becomes clear how these assurances can be invalidated.

To implement the above scheme, a few key items are required. First, a SoC featuring a secure boot mechanism is needed. This is an important factor to consider when evaluating potential components for a product. The security of the entire system ultimately rests upon the hardware-based root of trust, so this functionality is very much worth validating prior to committing to a particular family of parts.

Next, U-Boot must be configured to support the desired signature scheme and the "fitImage" format [107,108]. Unlike the legacy uImage format, the fitImage format can store multiple images in a single, cryptographically signed file. This provides a means to load and authenticate both a kernel and its accompanying metadata. The `kernel-fitimage` BitBake class [109], which inherits from `uboot-sign.bbclass` [110], can be used to produce signed fitImage binaries.

Finally, a means to authenticate the root filesystems contents is needed. This is much easier to achieve if the root filesystem is decoupled from mutable data and made read-only. Guidance on creating a read-only root filesystem is presented in [111,112]. Once your system is using a read-only root filesystem, it becomes possible to authenticate it using dm-verity [113–116]. This device-mapper (dm) target allows the integrity and authenticity of filesystem contents to be validated by constructing a hash tree from the data blocks on the underlying storage media, and then checking a cryptographic signature computed over the root-level hash.

Once executing trusted system software from the root filesystem, applications can manage the authenticity and confidentiality of mutable data stored on a separate writable partition. Furthermore, the system can manage and apply updates. For a comparison of available system update mechanisms, including their security features, see the Yocto Project wiki [117].

# 5 Preparing for Security Assessments

After spending months working on a system, it can be difficult to step back and perform an objective assessment of the result. This is just one reason why engaging a third-party to perform a security assessment can be beneficial. It is generally the case that a white-box assessment will yield the most value. Access to source code, build configurations, and existing test cases allows potential findings to be more efficiently tested and assessed within the limited time frame of an engagement. From the perspective of security consultants, having a more comprehensive view of a product also allows for more practical and actionable recommendations to be provided.

This section highlights some Yocto-specific items that are helpful to prepare for a security team prior to the start of a security assessment, and notes some of the potential benefits that can be gained by making this information available to reviewers.

## 5.1 Aggregating Build Metadata

As outlined in Section 2.2, a great deal of information about builds can be obtained by enabling a few configuration items. For the same reasons that this information allows you to assess the overall health and quality of builds, this can allow a security team to quickly spot "red flags". Therefore, it is suggested that the following items be included in the materials prepared for security assessment.

#### Build History

The items included in Table 1 can allow for an expedited determination of whether the software deployed on a system is up-to-date with security patches. Furthermore, information about files deployed in an image can also allow permissions-related defects and unintentionally deployed items to be more easily identified.

#### Layers

Access to the Yocto and OpenEmbedded layers used to build firmware images can allow a security team to better understand the origins of deployed files and the configuration options with which applications are built. If layers are provided as complete version control repositories, then the history information can be used to further understand the intent of changes, and sometimes better determine the root causes of both isolated and systemic issues.

#### Existing QA Tests and Prior Findings

Providing a third-party testing team with build-time tests (Section 3.3) that your organization uses internally, along with prior findings, can help jumpstart a security assessment and avoid duplicated efforts. This can also allow for a more holistic review of your organization's security processes. Gaps contributing to identified security vulnerabilities can more easily be identified, and opportunities to improve and expand your organization's QA processes can be explored.

## 5.2 Exporting an SDK

The OpenEmbedded build system used by Yocto allows image-specific SDKs to be built and exported [118,119]. These SDKs, typically bundled as a single self-extracting installer script, contain all of the tools and components needed to work with the software included in an image:

- A cross-compiler toolchain and debugger for a device's target architecture
- The libraries, header files, and symbols in an image
- A setup script akin to `oe-init-build-env` that initializes the SDK environment

Providing security testing teams with access to this suite of tools can greatly reduce the time needed to assess security defects and demonstrate potential impacts through proof-of-concept (PoC) exploits.

The "Standard SDK" is the simplest to produce and work with, and will often suffice for security testing purposes. "The Extensible SDK" provides significantly more functionality that may not be necessary outside of development teams' workflows.

The following command builds the Standard SDK for a specific image.

```
bitbake -c populate_sdk <target_image>
```

**Listing 29:** Building a Standard SDK for an image

Once this completes, the self-extracting installer will be located in `${BUILDDIR}/tmp/deploy/sdk`, along with host and target manifest files, which list the packages included in system root directories (sysroots) installed by the SDK.

Running the installer script allows the SDK to be extracted to a user-defined location. At the completion of the installation, the script notes how to initialize the environment. An example of this, using a Standard SDK generated using the Poky Reference Distro, is shown below:

```
$ sudo ./tmp/deploy/sdk/poky-glibc-x86_64-ncc-demo-image-arm1176jzfshf-vfp-toolchain-2.5.sh

Poky (Yocto Project Reference Distro) SDK installer version 2.5
===============================================================
Enter target directory for SDK (default: /opt/poky/2.5):
You are about to install the SDK to "/opt/poky/2.5". Proceed[Y/n]?
Extracting SDK.................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment
setup script e.g.
    $ . /opt/poky/2.5/environment-setup-arm1176jzfshf-vfp-poky-linux-gnueabi

$ . /opt/poky/2.5/environment-setup-arm1176jzfshf-vfp-poky-linux-gnueabi

$ echo ${CC}
arm-poky-linux-gnueabi-gcc  -march=armv6 -mfpu=vfp -mfloat-abi=hard -mtune=arm1176jzf-s      ↩
-mfpu=vfp --sysroot=/opt/poky/2.5/sysroots/arm1176jzfshf-vfp-poky-linux-gnueabi              ↩
linux-gnueabi

$ echo ${GDB}
arm-poky-linux-gnueabi-gdb
```

**Listing 30:** Installation of a Standard SDK

Once the environment is sourced, a number of variables, such as `${CC}`, `${CXX}`, `${CROSS_COMPILE}`, and `${GDB}` are initialized to point to appropriate cross-toolchain items, which are added to the user's `${PATH}`. The SDK includes a `sysroots` directory containing system root directories for both the host tools and target-specific firmware. The latter contains a superset of what is deployed in the firmware image, including items from recipes' `${PN}-dev` and `${PN}-dbg` auxiliary packages.

Note that the source code associated with each recipe is also included in the `/usr/src` directory of the target sysroot. By generating the SDK at the same time firmware releases are built, this helps ensure that source code provided to security teams for review is consistent with the binaries deployed in the firmware. However, auxiliary data that is included with source repositories, such as build scripts and configuration files, are not present in this directory. There is still value in providing the complete source code repository contents to testers when conducting a white-box assessment.

## 5.3 Simulated Hardware Interfaces and QEMU Targets

Performing security assessments at regular project milestones, rather than a single post-integration milestone prior to production launch, can significantly reduce the cost of remediating defects. This is especially true when the defect is architectural, rather than implementation-specific.

In the context of embedded systems this can be challenging to accomplish due to software components' dependencies upon hardware that may still be undergoing design, prototyping, debugging, or manufacturing. Even when fully functional prototypes are available, they may be in high demand within an organization, leaving few available for use in an external security assessment.

However, with a little foresight in the beginning of a project, it is often possible to design with this type of testing in mind. By using standard Linux subsystems and hardware abstraction layers, many types of devices can be emulated, or at least sufficiently simulated, in order to support the development and testing of application code. Here are just a few examples:

- The `mtdram`, `nandsim`, and `block2mtd` tools [120] could be used to emulate the NOR and NAND flash devices upon which UBI volumes or flash-based filesystems reside.
- Electronic Control Units in a vehicular network could be simulated using a virtual CAN interface [121], the Linux `can-utils` [122], and Python's socket support for the `PF_CAN` protocol family [123].
- Devices accessed via simple character device-based interfaces (e.g. `spidev`, `i2c-dev`) can often use simple replacement kernel modules to emulate device's behavior. Alternatively, FUSE can be leveraged to implement such a simulator in user space [124,125].

Once individual software components can be tested independently of physical hardware, Yocto's emulator targets become significantly more valuable in QA and security testing. Even without physical hardware, individual components can be tested in a firmware image, for the same processor architecture, running atop of the QEMU emulator [126].

Builds for QEMU targets can be created by changing the `${MACHINE}` variable in `local.conf` to one of the following and then performing a build.

- `qemuarm`
- `qemuarm64`
- `qemumips`
- `qemumips64`
- `qemux86`
- `qemux86-64`
- `qemuppc`

The Yocto Project Quick Build guide demonstrates this process using an emulated x86 system [14]. The `runqemu` helper script can be used to boot an image using QEMU [127]. Although it may be possible to build and test an application natively on your development machine, using these emulated targets can often help identify build-specific and architecture-specific defects before physical hardware is available.

# 6 Conclusion

Building a secure product is inherently challenging, especially as the complexities of modern embedded systems and their underlying components are constantly increasing. Many product development teams find the pursuit of security to be frustrating and nebulous; there is no one single off-the-shelf solution, test suite, or methodology that can eliminate all threats to network infrastructures, sensitive data, intellectual property, and the privacy of end users. To quote Bruce Schneier, "*Security is a process, not a product*" [128]. It is about forward-thinking and managing risks. As Internet-connected technology continues to become increasingly pervasive in our daily lives, integrating security-focused thinking into all aspects of product design, implementation, and maintenance is essential.

In the past, firmware builds simply represented a point at which an engineering team could hand off artifacts to a QA testing team. While other areas of computing have begun fostering better coordination and integration of development, deployment, and security teams,[20] the embedded computing application domains appears to have been slow to adopt such practices.

However, with the emergence and growing adoption of projects such as Yocto and OpenEmbedded, there is a significant opportunity for embedded systems teams to catch up. As shown in this paper, these tools can be used to not only monitor and inspect the health of firmware, but also to integrate and enforce security best practices into the build process itself. Through increased coordination between development and QA teams, automated checks can be leveraged to detect and prevent security defects earlier in project schedule, when remediation efforts are less costly. Readily available features and functionality within the Yocto ecosystem can be used to make systems more resilient to threats, as well as to prepare the resources necessary for more through and productive security reviews. Armed with these tools, product development and testing teams can be better prepared to deal with the security challenges that lie ahead.

[20]The term "SecDevOps" is often used to refer to this approach.

# Appendices

## A Version Control Best Practices

As noted in Section 2.1, disciplined use of version control plays an important role in maintaining visibility into how software is changing over time. Below are a few version control best practices that can help improve general code quality, which in turn can yield an improved security posture. Although these guidelines are specifically tailored to `git`, the underlying motivations and principles in each of these can often be applied to other version control software.

**Do not merge incomplete, non-functional, untested, or unreviewed code into the main source repository branch.**
Instead, develop functionality in topic-specific branches and gate their inclusion into the main codebase, pending adequate QA. The primary goal of this guideline is to integrate regular QA checkpoints into the version control workflow, thereby reducing the deployment of defective software. Within the context of performing security assessments, this helps ensure that the focus is on production-ready functionality, thereby reducing false positives due to some components being in a development or diagnostic configuration. See [129,130] for information on how this type of workflow can be achieved with `git`.

**Apply larger changes as a series of buildable, testable, small, and tightly-coupled commits.**
Smaller, tightly-coupled change sets are easier to understand and review, in comparison to massive changes spanning many components. When an entire codebase still builds and runs successfully with each successive commit, it becomes easier to isolate defects to specific changes via manual or automated bisection [131] of the version control history. This guideline is easiest to follow when committing many small changes to a local development branch, and then "squashing" commits [132] in a separate local staging branch prior to pushing them to an upstream repository.

**Write commit messages that clearly articulate the context of changes.**
Be sure to include details such as requirements or issue tracker items that are associated with changes. Do not think of version control as just a means to back up files. Treat it as a means to traverse and inspect the evolution of a project, as well as return to past states. Remember that the person inspecting version control history may be doing so months or years after any particular change; focus on the "*why*" and "*what*" rather than the "*how*" in the commit message body. See [133] for an excellent discussion about how to write useful commit messages.

**Do not store credentials or key material in version control repositories**.
This is even more important when using third-party hosting sites. Consider the impact of a version control repository compromise when deciding what information can be stored within it. A leak of source code should not undermine the security of a system or product. Restrict access to sensitive credentials or key material to team members on a need-to-know basis and load this information externally via environment variables or configuration files. Leverage hooks [134,135], or clean/smudge filters [136] to prevent this information from accidentally entering source code and layer repositories.

## B Introduction to Exploitation Mitigations

One way in which applications can be hardened is through the use of certain compiler features commonly referred to as "exploit mitigations." These features, often enabled through the use of additional compiler or linker flags, increase the difficulty of successfully exploiting certain types of security vulnerabilities. With exploit mitigations enabled, an attacker often needs to identify and successfully exploit *multiple* vulnerabilities of different defect classes in order to achieve the same end goal. As a result, one particular vulnerability might be rendered impractical to exploit reliably within a given timeframe or on a large scale.

Below are a few examples of exploit mitigations supported by popular compilers including GCC and Clang. While the following list is by no means exhaustive, it represents a number of exploit mitigations that modern Linux distributions enable by default. Section 3.2 describes how to identify if these mitigations are enabled, and Section 4.3 details how their corresponding compiler flags can be enabled in BitBake recipes.

### Stack Protection

Stack protection is a defense against stack-based buffer overflow attacks [137] that terminate a program when stack corruption is detected. The idea behind this strategy is that the controlled termination of the program is regarded as being safer than allowing an attacker to write arbitrary data to the stack, potentially directing a function's return address to a location containing malicious code. This assumes that abrupt program termination does not pose a serious risk. This assumption may not be appropriate for automotive, medical, or industrial control systems. Nonetheless, an out-of-bounds memory access that does not trigger immediate termination of a program is likely to result in unsafe behavior later in the program.

Stack protections are usually implemented by writing a pseudorandom value (referred to as a "canary" or "cookie") to a stack frame when a function is invoked, and then confirming that this value has not changed before returning from that function. If the pseudorandom value is detected to have been corrupted, the program is aborted, often by way of a `__stack_chk_fail()` function. When stack protections are enabled, the compiler introduces these checks automatically. However, because these added checks introduce overhead to functions, the degree to which they are applied throughout a codebase is typically a configurable option.

### Non-Executable Stack

The traditional technique for exploiting stack-based memory corruption vulnerabilities is to place a malicious executable payload on the stack and overwrite a function's return address to point to the location of this payload [138]. By configuring stack pages to be non-executable, this specific methodology is no longer possible. However other techniques, such as return-oriented programming [139], may still be possible.

### FORTIFY_SOURCE

GCC and Clang support a `_FORTIFY_SOURCE` macro to enable additional compile-time and runtime checks in libc functions [140]. Many of these additional checks aim to prevent these functions from accessing memory beyond the bounds of an associated buffer. In situations when an invalid access is detected, the program is terminated. Note that `_FORTIFY_SOURCE=2` enables all available runtime checking, which is often desirable from a security perspective.

### Address Space Layout Randomization

When an application is compiled as a Position Independent Executable, it is able to take advantage of the Address Space Layout Randomization (ASLR) functionality provided by the kernel, if this is enabled.[21] This complicates exploitation because the locations of code and data are not deterministic across successive executions of a program. As a result, an attacker would need to find an additional defect, such as a leak of address space information, before being able to develop a reliable exploit.

[21] See the documentation for `kernel.randomize_va_space=2` [141]

## Full RELocation Read-Only

When a program begins executing, the locations of dynamically loaded library functions may have not necessarily been resolved yet. When employing a “lazy loading” approach, functions are resolved the first time they are invoked. This requires that a certain portions of a process’s data sections[22] are writeable. An attacker can leverage these writable data sections to redirect program flow to malicious code, provided a sufficient memory corruption defect and some knowledge of a target process’s memory layout. In addition to re-ordering data sections to further reduce the risk of data sections becoming corrupted, Full RELocation Read-Only (RELRO) (re)maps the data sections and GOT as being read-only. This security benefit comes with a trade-off, however. Because all dynamically loaded symbols must be resolved upfront, start-up time will be slightly increased.

## Treating Format String Warnings as Errors

Incorrect usage of functions that handle format string specifiers can provide an attacker with an ability to both read and write to arbitrary memory locations. Elevating compiler warnings about incorrect or unsafe usages of format string functions to an error can help identify potentially dangerous code that requires changes. Although not technically an “exploit mitigation”, this has been included here to highlight the ability to enable stricter build processes through the use of additional compiler flags.

[22]The Procedure Linkage Table (PLT) and Global Offset Table (GOT)

## C Dependency Visualization Example

The following Python script implements the dependency graph visualization discussed in Section 2.2. It also available online: https://github.com/nccgroup/yocto-whitepaper-examples

```
#!/usr/bin/env python3
#
# SPDX-License-Identifier: MIT
# Author: Jon Szymaniak <jon.szymaniak.foss@gmail.com>

"""
Extract a subset of a dependency graph from a Yocto build history dot file.
"""

import argparse
import re
import sys

# Prepare command-line arguments
def handle_cmdline():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('-i', '--infile',   metavar='filename', required=True,
        help='Input DOT file from buildhistory')
    parser.add_argument('-o', '--outfile',  metavar='filename', required=True,
        help='Output DOT file containing graph subset')
    parser.add_argument('-p', '--pkg',      metavar='package',  required=True,
        help='Target package name')
    parser.add_argument('-H', '--height',   metavar='height',   type=int, default='3',
        help='# of dependency levels from the target package upward to include. Default: 3')
    parser.add_argument('-D', '--depth',    metavar='depth',    type=int, default='0',
        help='# of dependency levels from the target package downward to include. Default: 0')
    return parser.parse_args()

# Update a package dependency graph to include the specified "depends_on" and
# "required_by" relationships
def update_graph(graph, pkg, dep, attr):
    pkg_node = graph.get(pkg, { 'depends_on': [], 'required_by': [] })
    pkg_node['depends_on'].append((dep, attr))
    graph[pkg] = pkg_node

    dep_node = graph.get(dep, { 'depends_on': [], 'required_by': [] })
    dep_node['required_by'].append((pkg, None))
    graph[dep] = dep_node

# Load and parse a Yocto buildhistory dot file and construct the included
# package dependency graph
def load_file(filename):
    pkg = '[a-zA-Z0-9\-\./]+'             # Match package name and version
    attr = '(\s*(?P<attr>\[.*\]))?'       # Match graphviz edge attributes
    pat = '^"(?P<pkg>' + pkg + ')"\s*->\s*"(?P<dep>' + pkg + ')"' + attr + '$'
    regexp = re.compile(pat)

    graph = {}
    with open(filename, 'r') as infile:
        for line in infile:
            m = regexp.match(line)
            if m:
                update_graph(graph, m.group('pkg'), m.group('dep'), m.group('attr'))
    return graph

# Traverse a dependency graph, up to the specified depth, and collect a set of
# all encountered packages in the `collected` parameter
def collect_packages(g, packages, depth, relationship, collected):
    if depth < 0:
        return []

    to_visit = []
    for p in packages:
        collected.add(p)
        to_visit += [node[0] for node in g[p][relationship]]

    collect_packages(g, to_visit, depth-1, relationship, collected)

# Return string data in DOT syntax that represents a target package's
# dependency relationships for the specified number of levels.
def dot_data(graph, target, levels, relationship, color):
    ret = ''
    packages = set()
    collect_packages(graph, [target], levels, relationship, packages)
    for pkg in packages:
        if pkg != target:
            ret += '  "{:s}" [color={:s}, style=filled]\n'.format(pkg, color)
        for info in [dep for dep in graph[pkg]['depends_on'] if dep[0] in packages]:
            ret += '  "{:s}" -> "{:s}" {:s}\n'.format(pkg, info[0], info[1] or '')
    return ret

if __name__ == '__main__':
    args = handle_cmdline()
    graph = load_file(args.infile)

    data  = 'digraph G {\n'
    data += '  node [shape=box]\n'
    data += '  edge [fontsize=9]\n'
    data += '  "{:s}" [color=seagreen1, style=filled]\n'.format(args.pkg)
    try:
        data += dot_data(graph, args.pkg, args.height, 'required_by', 'lightblue')
        data += dot_data(graph, args.pkg, args.depth, 'depends_on', 'lightgray')

        data += '}\n'
        with open(args.outfile, 'w') as outfile:
            outfile.write(data)

    except KeyError:
        print('No such package in graph: ' + args.pkg, file=sys.stderr)
```

## D Example buck-security output

The following listing presents a buck-security report generated by a scan of a default `core-image-sato` build for a Raspberry Pi Zero. See Section 3.2.1 for the corresponding discussion of this output.

```
[1] CHECK emptypasswd: Users with empty password            [ WARNING ]
The security test discovered a possible insecurity.
The following users have empty passwords.
######################################################
root

[2] CHECK sgids: Files where Setgid is used                 [ OK ]

[3] CHECK stickytmp: Mode, user, and group acceptable for tmp directory. [ OK ]


[4] CHECK suids: Files where Setuid is used                 [ WARNING ]
The security test discovered a possible insecurity.
The following programs have the SUID set.
######################################################
Pathnames are relative to <snip>/core-image-sato/1.0-r0/rootfs.
/bin/busybox.suid
/bin/su.shadow
/sbin/halt.sysvinit
/sbin/shutdown.sysvinit
/usr/bin/apm
/usr/bin/chage
/usr/bin/chfn.shadow
/usr/bin/chsh.shadow
/usr/bin/expiry
/usr/bin/newgidmap
/usr/bin/newgrp.shadow
/usr/bin/newuidmap
/usr/bin/passwd.shadow
/usr/libexec/dbus-daemon-launch-helper

[5] CHECK superusers: Find superusers                       [ OK ]

[6] CHECK worldwriteablefiles: World Writeable Files        [ OK ]

[7] CHECK worldwriteabledirs: World Writeable Directories   [ WARNING ]
The security test discovered a possible insecurity.
The following directories are writeable for all users.
######################################################
Pathnames are relative to <snip>/core-image-sato/1.0-r0/rootfs.
/tmp
```

## E Example ISA FW Compile Flag Analyzer Report Snippets

The following listing presents selected snippets from the report generated by running the ISA FW Compile Flag Analyzer on a default `core-image-sato` build for a Raspberry Pi Zero. See Section 3.2.2 for the discussion of this report.

```
Report for image: core-image-sato
With rootfs location at
 ${BUILDDIR}/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/core-image-sato/1.0-r0/rootfs

Relocation Read-Only
More information about RELRO and how to enable it:
http://tk-blog.blogspot.de/2009/02/relro-not-so-well-known-memory.html
Files with no RELRO:
/lib/modules/4.14.39/kernel/fs/binfmt_misc.ko
/lib/modules/4.14.39/kernel/fs/jfs/jfs.ko
/lib/modules/4.14.39/kernel/fs/nfsd/nfsd.ko
...

Files with partial RELRO:
/usr/libexec/dbus-daemon-launch-helper
...
/usr/lib/libpcre.so.1.2.9
/usr/lib/libXtst.so.6
...
/usr/sbin/dropbear


Stack protection
More information about canary stack protection and how to enable it:
https://lwn.net/Articles/584225/
Files with no canary:
/usr/libexec/dbus-daemon-launch-helper
...
/usr/lib/libpcre.so.1.2.9
/usr/lib/libXtst.so.6
...
/usr/sbin/dropbear
...
/lib/modules/4.14.39/kernel/fs/binfmt_misc.ko
...


Position Independent Executable
More information about PIE protection and how to enable it:
https://securityblog.redhat.com/2012/11/28/position-independent-executables-pie
Files with no PIE:
/usr/libexec/dbus-daemon-launch-helper
...
/usr/lib/libpcre.so.1.2.9
/usr/lib/libXtst.so.6
...
/usr/sbin/dropbear
...


Non-executable stack
Files with executable stack enabled:
```

```
Files with no ability to fetch executable stack status:


Grop initialization:
If using setuid/setgid calls in code, one must call initgroups or setgroups
Files that don't initialize groups while using setuid/setgid:
/usr/libexec/sudo/sudoers.so
/usr/bin/Xorg
/usr/bin/X
/usr/bin/sudoedit
/usr/bin/sudo
/usr/sbin/ofonod
/lib/libmount.so.1.1.0
/bin/bash.bash


Memory Protection Extensions
More information about MPX protection and how to enable it:
https://software.intel.com/sites/default/files/
    managed/9d/f6/Intel_MPX_EnablingGuide.pdf
Files that don't have MPX protection enabled:
/var/lib/dnf/history/history-2018-06-09.sqlite
/usr/libexec/dbus-daemon-launch-helper
/usr/libexec/ck-get-x11-server-pid
...
```

## F Example ISA FW Filesystem Analyzer Report

The following listing presents the report generated by running the ISA FW Filesystem Analyzer on a default `core-image-sato` build for a Raspberry Pi Zero. See Section 3.2.2 for the discussion of this report.

```
Report for image: core-image-sato
With rootfs location at
  ${BUILDDIR}/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/core-image-sato/1.0-r0/rootfs

Files with SETUID bit set:
/usr/libexec/dbus-daemon-launch-helper
/usr/bin/gpasswd
/usr/bin/apm
/usr/bin/newuidmap
/usr/bin/expiry
/usr/bin/passwd.shadow
/usr/bin/chsh.shadow
/usr/bin/chage
/usr/bin/newgrp.shadow
/usr/bin/sudo
/usr/bin/newgidmap
/usr/bin/chfn.shadow
/bin/su.shadow
/bin/busybox.suid
/sbin/halt.sysvinit
/sbin/shutdown.sysvinit


Files with SETGID bit set:


World-writable files:


World-writable dirs with no sticky bit:
```

## G Example ISA FW Kernel Configuration Analyzer Report

The following listing presents just a few snippets from a report generated by running the ISA FW Kernel Configuration Checker on a default `core-image-sato` build for a Raspberry Pi Zero. See Section 3.2.2 for the discussion of this report.

```
Report for image: core-image-sato
With the kernel conf at:
  ${BUILDDIR}/tmp/work-shared/raspberrypi0-wifi/kernel-build-artifacts/.config

Hardening options that need improvement:

Actual value:
CONFIG_CC_STACKPROTECTOR : not set
Recommended value:
CONFIG_CC_STACKPROTECTOR : y
Comment: Enables the stack protector GCC feature which defends against
stack-based buffer overflows

Actual value:
CONFIG_CROSS_MEMORY_ATTACH : y
Recommended value:
CONFIG_CROSS_MEMORY_ATTACH : not set
Comment: Enables cross-process virtual memory access. Providing virtual memory
access to and from a hostile process would assist an attacker in discovering
attack vectors.

Actual value:
CONFIG_DEBUG_KERNEL : y
Recommended value:
CONFIG_DEBUG_KERNEL : not set
Comment: Enables sysfs output intended to assist with debugging a kernel. The
information output to sysfs would assist an attacker in discovering attack
vectors.

Actual value:
CONFIG_DEBUG_RODATA : not set
Recommended value:
CONFIG_DEBUG_RODATA : y
Comment: Sets kernel text and rodata sections as read-only and write-protected.
This guards against malicious attempts to change the kernel's executable code.

Actual value:
CONFIG_DEBUG_STRICT_USER_COPY_CHECKS : not set
Recommended value:
CONFIG_DEBUG_STRICT_USER_COPY_CHECKS : y
Comment: Converts a certain set of sanity checks for user copy operations into
compile time failures. The copy_from_user() etc checks help test if there are
sufficient security checks on the length argument of the copy operation by
having gcc prove that the argument is within bounds.

Actual value:
CONFIG_DEVMEM : y
Recommended value:
CONFIG_DEVMEM : not set
Comment: Enables mem device, which provides access to physical memory.
Providing a view into physical memory would assist an attacker in discovering
attack vectors.
```

```
Actual value:
CONFIG_FTRACE : y
Recommended value:
CONFIG_FTRACE : not set
Comment: Enables the kernel to trace every function. Providing kernel trace
functionality would assist an attacker in discovering attack vectors.

Actual value:
CONFIG_IKCONFIG_PROC : y
Recommended value:
CONFIG_IKCONFIG_PROC : not set
Comment: Enables access to the kernel config through /proc/config.gz. Leaking
the kernel configuration would assist an attacker in discovering attack
vectors.

Actual value:
CONFIG_SERIAL_CORE : y
Recommended value:
CONFIG_SERIAL_CORE : not set
Comment: Enables the serial console. Providing access to the serial console
would assist an attacker in discovering attack vectors.

...

Key-related options that need improvement:

Actual value:
CONFIG_ENCRYPTED_KEYS : not set
Recommended value:
CONFIG_ENCRYPTED_KEYS : y

Actual value:
CONFIG_TRUSTED_KEYS : not set
Recommended value:
CONFIG_TRUSTED_KEYS : y

Security options that need improvement:

Actual value:
CONFIG_DEFAULT_SECURITY : ""
Recommended value:
CONFIG_DEFAULT_SECURITY : "selinux","smack","apparmor","tomoyo"

...
```

## H rootfs-file-blacklist.bbclass

The following BitBake class implements the file blacklist functionality detailed in Section 3.3.1. It also available online: https://github.com/nccgroup/yocto-whitepaper-examples

```
# Search for blacklisted files deployed in a root filesystem, as defined by
# their name or type (according to the program 'file').
#
# SPDX-License-Identifier: MIT
# Author: Jon Szymaniak <jon.szymaniak.foss@gmail.com>

# Case-insensitive and semicolon-delimited list of filenames to blacklist.
# Glob-style wildcards (e.g. *.pem) are permitted.
BLACKLISTED_FILE_NAMES ?= ""

# Case insensitive and semicolon-delimited list of file types,
# as defined by the output of the "file" program
BLACKLISTED_FILE_TYPES ?= ""

python find_blacklisted_files() {
    from fnmatch import fnmatch

    type_blacklist = set()
    name_blacklist = set()
    glob_blacklist = []

    for entry in d.getVar('BLACKLISTED_FILE_NAMES').split(';'):
        entry = entry.strip().lower()
        if len(entry) == 0:
            continue
        elif '*' in entry:
            glob_blacklist.append(entry)
        else:
            name_blacklist.add(entry)

    for entry in d.getVar('BLACKLISTED_FILE_TYPES').split(';'):
        entry = entry.strip().lower()
        if len(entry):
            type_blacklist.add(entry)

    rootfs = d.getVar('IMAGE_ROOTFS')
    for root, dirs, files in os.walk(rootfs):
        for f in files:
            curr_path = os.path.join(root, f)
            dest_path = os.path.join(root[len(rootfs):], f)
            f = f.lower()

            if f in name_blacklist or any(fnmatch(f, g) for g in glob_blacklist):
                bb.warn('Blacklisted file (name) in rootfs: ' + dest_path)
            else:
                output, err = bb.process.run('file -b ' + curr_path)
                if any(t in output.lower() for t in type_blacklist):
                    bb.warn('Blacklisted file (type) in rootfs: ' + dest_path)
}

ROOTFS_POSTPROCESS_COMMAND += "find_blacklisted_files;"
DEPENDS += "file-native"
```

## I symbol-blacklist.bbclass

The following BitBake class implements the symbol blacklist functionality detailed in Section 3.3.2. It also available online: https://github.com/nccgroup/yocto-whitepaper-examples

```
# Blacklist symbols within applications and libraries
#
# SPDX-License-Identifier: MIT
# Author: Jon Szymaniak <jon.szymaniak.foss@gmail.com>

# Space-delimited list of blacklisted symbol names
BLACKLISTED_SYMBOLS ?= "\
    atoi atol atof \
    gets \
    strcpy stpcpy strcat \
    sprintf vsprintf \
"

def is_elf(f):
    out, err = bb.process.run('file -b ' + f)
    return 'ELF' in out

def check_file(d, file_path):
    if not is_elf(file_path):
        return

    readelf = d.getVar('READELF')
    out, err = bb.process.run(readelf + ' --syms ' + file_path)
    for sym in d.getVar('BLACKLISTED_SYMBOLS').split():
        if sym in out:
            msg = 'Blacklisted symbol "{:s}" present in {:s}'
            bb.warn(msg.format(sym, file_path))

python do_find_blacklisted_symbols() {
    install_dir = d.getVar('D')
    for root, dirs, files in os.walk(install_dir):
        for f in files:
            check_file(d, os.path.join(root, f))
}

addtask do_find_blacklisted_symbols after do_install before do_package
```

# References

[1] https://www.openembedded.org. 4

[2] https://www.yoctoproject.org. 4

[3] https://buildroot.org. 4

[4] A. Belloni and T. Petazzoni, “Buildroot vs OpenEmbedded/Yocto: A Four Hands Discussion.” Embedded Linux Conference. 2016. https://www.youtube.com/watch?v=13LZ0szWSVg. 4

[5] https://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/about. 4

[6] https://git.yoctoproject.org/cgit/cgit.cgi/meta-fsl-arm/about. 4

[7] https://git.yoctoproject.org/cgit/cgit.cgi/meta-freescale/about. 4

[8] https://git.yoctoproject.org/cgit.cgi/meta-xilinx/about. 4

[9] https://github.com/kraj/meta-altera. 4

[10] G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, May 2002. RTI Project Number 7007.011. https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf. 4

[11] “Docs Overview.” Yocto Project. https://www.yoctoproject.org/docs. 5

[12] R. Purdie, C. Larson, and P. Blundell, *BitBake User Manual*. BitBake Community. https://www.yoctoproject.org/docs/2.5/bitbake-user-manual/bitbake-user-manual.html. 5, 17

[13] S. Rifenbark, *Yocto Project Reference Manual*. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html. 5

[14] “Yocto Project Quick Build.” https://www.yoctoproject.org/docs/2.5/brief-yoctoprojectqs/brief-yoctoprojectqs.html. 5, 36

[15] J. Szymaniak, “Ready, Set, Yocto!.” Revision 2.5. https://github.com/jynik/ready-set-yocto. 5

[16] https://www.raspberrypi.org/products/raspberry-pi-zero. 5

[17] S. Rifenbark, *Yocto Project Overview and Concepts Manual*, ch. 3.5. Git. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#git. 6

[18] Atlassian, “What is version control?.” https://confluence.atlassian.com/get-started-with-bitbucket/what-is-version-control-856845190.html. 6

[19] “GitHub Learning Lab.” https://lab.github.com. 6

[20] S. Chacon and B. Straub, *Pro Git*. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2. 6

[21] Mercurial Wiki, “Beginner’s Guides.” https://www.mercurial-scm.org/wiki/BeginnersGuides. 6

[22] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 3.28. Maintaining Build Output Quality. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#maintaining-build-output-quality. 6

[23] https://graphviz.gitlab.io/_pages/doc/info/lang.html. 8

[24] https://www.graphviz.org/. 8

[25] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 3.1.5. Using .bbappend Files in Your Layer. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#using-bbappend-files. 9, 28

[26] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 3.28.1. Enabling and Disabling Build History. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#enabling-and-disabling-build-history. 9

[27] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 3.28.2.5. Examining Build History Information. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#examining-build-history-information. 10

[28] https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/classes/cve-check.bbclass. 11

[29] “Archived Releases. Release Information - YP Core Morty 2.2 - 2016.10.28.” Yocto Project. https://www.yoctoproject.org/software-overview/downloads/archived-releases. 11

[30] https://nvd.nist.gov. 11

[31] https://cve.mitre.org. 11

[32] M. Vanhoef and F. Piessens, “Key reinstallation attacks: Forcing nonce reuse in WPA2,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2017. https://www.krackattacks.com. 11

[33] https://git.yoctoproject.org/cgit/cgit.cgi/meta-security/about. 13

[34] https://github.com/01org/meta-security-isafw. 13

[35] http://www.buck-security.net/documentation.html. 13

[36] https://github.com/davewood/buck-security. 13

[37] S. Rifenbark, *Yocto Project Reference Manual*, ch. 13. Variables Glossary, ROOTFS_POSPROCESS_COMMMAND. Linux Foundation, May 2018. Revision 2.5.
https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#var-ROOTFS_POSTPROCESS_COMMAND. 13

[38] S. Rifenbark, *Yocto Project Reference Manual*, ch. 12. Features. Linux Foundation, May 2018. Revision 2.5.
https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-features. 14, 28

[39] M. Kerrisk, *The Linux Programming Interface*, ch. 38 Writing Secure Priviledged Programs. No Starch Press, 2010.
http://man7.org/tlpi. 14, 25

[40] A. Grattafiori, *Understanding and Hardening Linux Containers*. NCC Group, April 2016.
https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers. 14, 21

[41] J. Hertz, *Abusing Privileged and Unprivileged Linux Containers*. NCC Group, June 2016.
https://www.nccgroup.trust/us/our-research/abusing-privileged-and-unprivileged-linux-containers. 14, 21

[42] M. Kerrisk, *The Linux Programming Interface*, ch. 39 Capabilities. No Starch Press, 2010.
http://man7.org/tlpi. 14, 26

[43] *The Linux kernel user-space API Guide*. Seccomp BPF.
https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. 14, 26

[44] *The Linux kernel user's and administrator's guide*. Linux Security Module Usage.
https://www.kernel.org/doc/html/latest/admin-guide/LSM/index.html. 14

[45] https://github.com/slimm609/checksec.sh. 14, 17

[46] https://tracker.debian.org/pkg/checksecurity. 14

[47] https://cisofy.com/lynis/. 14

[48] https://www.open-scap.org/. 14

[49] https://github.com/01org/isafw. 14

[50] http://www.trapkit.de/tools/checksec.html. 15

[51] http://man7.org/linux/man-pages/man8/execstack.8.html. 15

[52] https://sourceware.org/binutils/docs/binutils/readelf.html. 15

[53] https://sourceware.org/binutils/docs/binutils/objdump.html. 15

[54] R. "ElfMaster" O'Neil, "RelroS: Read Only Relocations for Static ELF," in *PoC||GTFO 0x18*, June 2018.
https://www.alchemistowl.org/pocorgtfo/pocorgtfo18.pdf. 15

[55] S. Rifenbark, *Yocto Project Reference Manual*, ch. 6.57. insane.bbclass. Linux Foundation, May 2018. Revision 2.5.
https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-classes-insane. 17

[56] S. Rifenbark, *Yocto Project Reference Manual*, ch. 13. Variables Glossary, IMAGE_ROOTFS. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#var-IMAGE_ROOTFS. 18

[57] "Security - ArchWiki." https://wiki.archlinux.org/index.php/security. 21

[58] "Hardening - Debian Wiki." https://wiki.debian.org/Hardening. 21

[59] A. Reelsen and J. F. S. Peña, "Securing Debian Manual," April 2012.
https://www.debian.org/doc/manuals/securing-debian-howto. 21

[60] "Introduction to Hardened Gentoo." https://wiki.gentoo.org/wiki/Hardened/Introduction_to_Hardened_Gentoo. 21

[61] "Gentoo Security Handbook." https://wiki.gentoo.org/wiki/Security_Handbook. 21

[62] J. Fuller, J. Ha, D. O'Brien, S. Radvan, E. Christensen, A. Ligas, M. McAllister, D. Walsh, D. Grift, E. Paris, and J. Morris, "Fedora Security Guide." https://docs.fedoraproject.org/en-US/Fedora/19/html/Security_Guide/index.html. 21

[63] "AGL Security Blueprint: System Hardening."
http://docs.automotivelinux.org/docs/architecture/en/dev/reference/security/07-system-hardening.html. 21, 30

[64] S. Rifenbark, *Yocto Project Overview and Concepts Manual*, ch. 2.5. Reference Embedded Distribution (Poky). Linux Foundation, May 2018. Revision 2.5.
https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html#reference-embedded-distribution. 21

[65] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 3.19. Creating Your Own Distribution. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#creating-your-own-distribution. 22

[66] https://github.com/Angstrom-distribution/meta-angstrom/blob/master/conf/distro/angstrom.conf. 22

[67] S. Rifenbark, *Yocto Project Reference Manual*, ch. 13. Variables Glossary. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html. 22

[68] S. Rifenbark, *Yocto Project Reference Manual*, ch. 6.88 own-mirrors.bbclass. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-classes-own-mirrors. 22

[69] S. Rifenbark, *Yocto Project Support Package (BSP) Developer's Guide*, ch. 1.8.2. BSP Machine Configuration Example. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/bsp-guide/bsp-guide.html#bsp-machine-configuration-example. 23

[70] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, pp. 388–402, July 1974. 25

[71] OWASP, "Command Injection." https://www.owasp.org/index.php/Command_Injection. 25, 29

[72] OWASP, "Cross-Site Request Forgery (CSRF)." https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF). 25

[73] M. Warner and J. Makinen, *Technical Advisory: Command Injection and CSRF in Quantenna Chip Affecting Multiple Networking Devices*. NCC Group, April 2017. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2017/april/technical-advisory-quentanna. 25, 29

[74] M. Kerrisk, "sockets/scm_cred_recv.c." from The Linux Programming Interface. http://man7.org/tlpi/code/online/dist/sockets/scm_cred_recv.c.html. 25

[75] S. Rifenbark, *Yocto Project Reference Manual*, ch. 6.144. useradd*.bbclass. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-classes-useradd. 26

[76] "getpwnam(3)." Linux Programmer's Manual. http://man7.org/linux/man-pages/man3/getpwnam.3.html. 26

[77] S. Rifenbark, *Yocto Project Reference Manual*, ch. 13. Variables Glossary, USERADDEXTENSION. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#var-USERADDEXTENSION. 26

[78] S. Rifenbark, *Yocto Project Reference Manual*, ch. 6.36 extrausers.bbclass. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-classes-extrausers. 27

[79] R. Wood, *Secure Device Manufacturing: Supply Chain Security Resilience*. NCC Group, September 2015. https://www.nccgroup.trust/uk/our-research/secure-device-manufacturing-supply-chain-security-resilience. 27

[80] S. Rifenbark, *Yocto Project Reference Manual*, ch. 14.2.4. Extra Build Information. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html#ref-varlocality-recipe-build. 28

[81] R. Purdie, C. Larson, and P. Blundell, *BitBake User Manual*, ch. 3.1.10. Removal (Override Style Syntax). BitBake Community. https://www.yoctoproject.org/docs/2.5/bitbake-user-manual/bitbake-user-manual.html#removing-override-style-syntax. 28

[82] S. Rifenbark, *Yocto Project Support Package (BSP) Developer's Guide*, ch. 1.6. Customizing a Recipe for a BSP. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/bsp-guide/bsp-guide.html#customizing-a-recipe-for-a-bsp. 28

[83] https://www.shodan.io/explore/category/industrial-control-systems. 29

[84] https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/recipes-extended/iptables/iptables_1.6.2.bb. 29

[85] https://busybox.net/about.html. 29

[86] https://grsecurity.net/features.php. 30

[87] "Kernel Self-Protection." The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/security/self-protection.html. 30

[88] https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project. 30

[89] M. Rutland, "Thrwarting Unknown Bugs: Hardening Features in the Mainline Linux Kernel." Embedded Linux Conference. 2016. https://www.youtube.com/watch?v=vuOZeZSAJc0. 30

[90] https://git.denx.de/?p=u-boot.git;a=commit;h=2fbb8462b0e18893b4b739705db047ffda82d4fc. 31

[91] B. Dixon, "Pin2Pwn: How to Root an Embedded Linux Box with a Sewing Needle." DEF CON 24. 2014. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Brad-Dixon-Pin2Pwn-How-to-Root-An-Embedded-Linux-Box-With-A-Sewing-Needle.pdf. 31

[92] Exploitee.rs Wiki, "Wink Hub: NAND Glitch Method." https://www.exploitee.rs/index.php/Wink_Hub%E2%80%8B%E2%80%8B#NAND_Glitch_Method_.28Works_on_any_Wink_Hub_FW.29. 31

[93] Exploitee.rs Wiki, "Staples Connect Hub: Exploitation." https://www.exploitee.rs/index.php/Staples_Connect_Hub%E2%80%8B%E2%80%8B#Exploitation. 31

[94] https://git.denx.de/?p=u-boot.git;a=blob;f=README;h=b1ddf89fc588290a85311c713260df3718e1c86e;hb=8c5d4fd0ec222701598a27b26ab7265d4cee45a3#l739. 31

[95] https://git.denx.de/?p=u-boot.git;a=blob;f=board/inversepath/usbarmory/usbarmory.c;h=a490aa814e75cb3cccbca8ee95312190e4c9fad6;hb=8c5d4fd0ec222701598a27b26ab7265d4cee45a3. 31

[96] https://git.denx.de/?p=u-boot.git;a=blob;f=env/Kconfig;h=8618376f252f68cbe30726b60dc64141d3521573;hb=8c5d4fd0ec222701598a27b26ab7265d4cee45a3#l3. 31

[97] https://git.denx.de/?p=u-boot.git;a=blob;f=env/nowhere.c;h=ea6c32eb3b77224eabf992cbd8ce333281ab4db4;hb=8c5d4fd0ec222701598a27b26ab7265d4cee45a3. 31

[98] "AppArmor Project Wiki." https://gitlab.com/apparmor/apparmor/wikis/home. 32

[99] "AppArmor Documentation." https://gitlab.com/apparmor/apparmor/wikis/Documentation. 32

[100] *The Linux kernel user's and administrator's guide*. Linux Security Module Usage: AppArmor. https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/apparmor.html. 32

[101] "SELinux Project Wiki." https://selinuxproject.org/page/Main_Page. 32

[102] https://github.com/SELinuxProject. 32

[103] *The Linux kernel user's and administrator's guide*. Linux Security Module Usage: SELinux. https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/SELinux.html. 32

[104] N. Elenkov, *Android Security Internals*, ch. 12. SELinux. No Starch Press, 2015. https://nostarch.com/androidsecurity. 32

[105] http://git.yoctoproject.org/cgit/cgit.cgi/meta-security/tree/recipes-security/AppArmor/apparmor_2.11.0.bb?h=sumo. 32

[106] https://git.yoctoproject.org/cgit/cgit.cgi/meta-selinux/about. 32

[107] M. Vašut, "Secure and flexible boot with U-Boot bootloader." Embedded Linux Conference Europe 2014. https://events.static.linuxfound.org/sites/events/files/slides/elce-2014.pdf. 33

[108] S. Glass, "Verified U-Boot," *LWN.net*, October 2013. https://lwn.net/Articles/571031. 33

[109] https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/classes/kernel-fitimage.bbclass?h=sumo. 33

[110] https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/classes/uboot-sign.bbclass?h=sumo. 33

[111] "Yocto Project Development Tasks Manual: 3.27. Creating a Read-Only Root Filesystem." https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#creating-a-read-only-root-filesystem. 33

[112] C. Simmonds, "Read-only rootfs: Theory and practice." Embedded Linux Conference Europe 2016. https://events.static.linuxfound.org/sites/events/files/slides/readonly-rootfs-elce-2016.pdf. 33

[113] "DMVerity. Cryptsetup wiki." https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity. 33

[114] "dm-verity." The Linux Kernel Documentation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/verity.txt. 33

[115] Android Security, "Verified Boot." https://source.android.com/security/verifiedboot. 33

[116] Android Security, "Implementing dm-verity." https://source.android.com/security/verifiedboot/dm-verity. 33

[117] Yocto Project Wiki, "System Update." https://wiki.yoctoproject.org/wiki/System_Update. 33

[118] S. Rifenbark, *Yocto Project Application Development and the Extensible Software Development Kit*. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/sdk-manual/sdk-manual.html. 34

[119] S. Rifenbark, *Yocto Project Application Development and the Extensible Software Development Kit*, ch. A.2. Building an SDK Installer. Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/sdk-manual/sdk-manual.html#sdk-building-an-sdk-installer. 34

[120] "UBI FAQ and HOWTO: How do I debug UBI?." http://www.linux-mtd.infradead.org/faq/ubi.html. 36

[121] *Linux SocketCAN Documentation*, ch. 6.4: The virtual CAN driver. https://www.kernel.org/doc/Documentation/networking/can.txt. 36

[122] "Linux-CAN / SocketCAN user space applications." https://github.com/linux-can/can-utils. 36

[123] "What's New In Python 3.3 - Python 3.7.0 Documentation. socket." https://docs.python.org/3/whatsnew/3.3.html?highlight=socketcan#socket. 36

[124] Codelectron, "How to setup virtual SPI in Linux." http://codelectron.com/how-to-setup-virtual-spi-in-linux/. 36

[125] https://github.com/codelectron/virtualSPI. 36

[126] https://www.qemu.org. 36

[127] S. Rifenbark, *Yocto Project Development Tasks Manual*, ch. 4. Using the Quick EMUlator (QEMU). Linux Foundation, May 2018. Revision 2.5. https://www.yoctoproject.org/docs/2.5/dev-manual/dev-manual.html#dev-manual-qemu. 36

[128] B. Schneier, "The Process of Security," *Information Security*, April 2000. https://www.schneier.com/essays/archives/2000/04/the_process_of_secur.html. 37

[129] S. Chacon and B. Straub, *Pro Git*, ch. 3.4 Git Branching - Branching Workflows. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows. 38

[130] S. Chacon and B. Straub, *Pro Git*, ch. 5.1 Distributed Git - Distributed Workflows. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows. 38

[131] S. Chacon and B. Straub, *Pro Git*, ch. 7.10 Git Tools - Debugging with Git, Binary Search. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Git-Tools-Debugging-with-Git. 38

[132] S. Chacon and B. Straub, *Pro Git*, ch. 7.6 Git Tools - Rewriting History. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History. 38

[133] C. Beams, "How to Write a Git Commit Message," August 2014. https://chris.beams.io/posts/git-commit. 38

[134] S. Chacon and B. Straub, *Pro Git*, ch. 8.3 Customizing Git - Git Hooks. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks. 38

[135] https://github.com/awslabs/git-secrets. 38

[136] S. Chacon and B. Straub, *Pro Git*, ch. 8.2 Customizing Git - Git Attributes. Apress, 2 ed., 2014. https://git-scm.com/book/en/v2/Customizing-Git-Git-Attributes. 38

[137] Robert C. Seacord, *Secure Coding in C and C++*, ch. 2.3 String Vulnerabilities and Exploits, pp. 53–69. Addison Wesley, 2 ed., 2009. 39

[138] Aleph One, "Smashing the Stack for Fun and Profit," in *Phrack Volume 7, Issue Forty-Nine*, 1996. http://phrack.org/issues/49/14.html. 39

[139] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS 2007* (S. De Capitani di Vimercati and P. Syverson, eds.), pp. 552–61, ACM Press, Oct. 2007. http://cseweb.ucsd.edu/~hovav/papers/s07.html. 39

[140] "FEATURE_TEST_MACROS(7)." Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/feature_test_macros.7.html. 39

[141] "/proc/sys/kernel: randomize_va_space." The Linux Kernel Documentation. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/Documentation/sysctl/kernel.txt?h=v4.16.8#n798. 39