Document Title	Requirements on Security Management for Adaptive Platform
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	881

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	17-10

Document Change History			
Date	**Release**	**Changed by**	**Description**
2017-10-27	17-10	AUTOSAR Release Management	• Initial Release

Disclaimer

Table of Contents

Bibliography

[1] Glossary
AUTOSAR_TR_Glossary

[2] Main Requirements
AUTOSAR_RS_Main

[3] Explanation of Adaptive Platform Design
AUTOSAR_EXP_PlatformDesign

[4] SoK: Eternal War in Memory

[5] Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR_RS_CPP14Guidelines

[6] The SPARC Architectural Manual, Version 8
http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz

[7] OpenBSD-3.3 announcement, public release of WX
http://www.openbsd.org/33.html

[8] Smashing The Stack For Fun And Profit
http://phrack.org/issues/49/14.html

[9] The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

[10] Jump-oriented Programming: A New Class of Code-reuse Attack

[11] On the Expressiveness of Return-into-libc Attacks

[12] Code-Pointer Integrity

[13] ARM Pointer Authentication
https://lwn.net/Articles/718888/

[14] PaX ASLR (Address Space Layout Randomization)

[15] Control-flow Integrity

[16] AMD64 Architecture Programmer's Manual Volume 2: System Programming
http://support.amd.com/TechDocs/24593.pdf

[17] PowerPC Architecture Book, Version 2.02
https://www.ibm.com/developerworks/systems/library/es-archguide-v2.html

[18] PowerPC Operating Environment Architecture Book III
http://public.dhe.ibm.com/software/dw/library/es-ppcbook3.zip

[19] Linux Kernel, Summary of changes from v2.6.7 to v2.6.8
https://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.8

[20] PAX
https://pax.grsecurity.net/docs/pax.txt

[21] CPI LLVM on github
https://github.com/cpi-llvm

[22] Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors

[23] Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

[24] ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks

[25] A seccomp overview
https://lwn.net/Articles/656307/

[26] Frequently Asked Questions for FreeBSD 10.X and 11.X

[27] pledge(2)
https://man.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2

1 Scope of Document

This document specifies the requirements of Adaptive Applications to the functional cluster Security of the AUTOSAR Adaptive Platform. The motivation is to provide standardized and portable security in Adaptive Applications.

2 Acronyms and abbreviations

The glossary below includes acronyms and abbreviations relevant to AP_RS_SecurityManagement that are not included in the AUTOSAR Glossary [1].

Abbreviation / Acronym:	Description:
Access Control Policy	This is an artifact which holds the information defining the basis for access control decisions. The necessary information is stored in the application manifest files which are distributed during provisioning.
Access Control Decision	The access control decision is a Boolean value indicating if the requested operation is permitted or not. The access control decision is based on access control policies.
Policy Decision Point (PDP)	This component makes the access control decision for a request based on the access control policy. It determines if the application is allowed to perform the requested task.
Policy Enforcement Point (PEP)	This component enforces the access control decision that is provided by the PDP. It communicates directly with the PDP to receive the access control decision.
Identity and Access Management (IAM)	Is a framework designed to be uniformly applicable to implementers of the AUTOSAR AP. The framework is about managing access rights to AUTOSAR entities.
Identity Information	Consists of an Application ID, Application instance ID, and their corresponding capabilities
Capability	A capability is a property of an AUTOSAR Application. By having a capability the AUTOSAR Application explicitly possesses the right to access resources/functionality that require this capability. Typical examples for a capability are: • Providing a specific service interface • Having access to a provided interface • Calling a specific interface in the functional cluster A capability is the explicit documentation of rights of an Adaptive Application for the IAM framework.

Table 2.1: Acronyms and Abbreviations

3 Requirements Specification

3.1 Functional Overview

The AUTOSAR Adaptive Platform Security provides services for Adaptive Applications and other clusters of the AUTOSAR Adaptive Platform. The AUTOSAR Adaptive Platform Security is responsible for all aspects which regards to:

- Crypto Stack
- Identity and Access Management
- Secure Communication
- Protected Runtime Environment

3.2 Identity and Access Management

[RS_SEC_03001] Limited access to adaptive platform services or APIs ⌈

Type:	draft
Description:	An adaptive platform instance shall provide means to limit application access to platform services or APIs.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00514)*

[RS_SEC_03002] Access control policies shall be enforced ⌈

Type:	draft
Description:	Access control policies shall be enforced during startup of adaptive applications that need access control.
Rationale:	Applications that depend on access control during startup have to covered by IAM. Therefore IAM should be available as soon as possible.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00514)*

[RS_SEC_03003] Access control shall be enforced by the `Adaptive Platform Foundation` ⌈

Type:	draft
Description:	Access control shall be enforced by the `Adaptive Platform Foundation` and not by the application.

Rationale:	Applications are considered to potentially being compromised.
Dependencies:	RS_SEC_03004
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03004] Applications shall be prevented from taking control over the PEP and PDP ⌈

Type:	draft
Description:	Applications shall be prevented from taking control over the PEP and PDP components of the IAM framework.
Rationale:	Appropriate measures have to be implemented. A working protected runtime environment shall be in place.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03005] Applications shall not circumvent PEPs ⌈

Type:	draft
Description:	Applications must be prevented from circumventing PEPs by using other APIs than the AUTOSAR defined APIs.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03006] Prevent applications from gaining information about the permissions of other applications ⌈

Type:	draft
Description:	The `Adaptive Platform Foundation` shall prevent applications from gaining information about the permissions of other applications.
Rationale:	Applications should not be able to gather information about the access rights of other applications.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03007] Access control policies shall be available to the PDP ⌈

Type:	draft

Description:	Access control policies shall be available to the PDP.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03008] `Adaptive Platform Foundation` shall provide access control decisions ⌈

Type:	draft
Description:	The adaptive `Adaptive Platform Foundation` shall provide access control decisions based on access control policies and capabilities that are stored in the corresponding manifests.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03009] Adaptive applications shall only be able to access persistent data when authorized ⌈

Type:	draft
Description:	`Adaptive Platform Foundation` shall ensure that adaptive applications are only able to access persistent data if a configured policy authorizes them to do so.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00330*, *RS_Main_00514*)

[RS_SEC_03010] Adaptive applications shall only be able to use service interfaces when authorized ⌈

Type:	draft
Description:	`Adaptive Platform Foundation` shall ensure that adaptive applications are only able to use a service interface if a configured policy authorizes them to do so.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00060*, *RS_Main_00514*)

[RS_SEC_03011] Policies shall be enforced by the local `Adaptive Platform Foundation` ⌈

Type:	draft
Description:	Policies shall be enforced by the local `Adaptive Platform Foundation`, i.e., that `Adaptive Platform Foundation` that runs on the machine of the requesting application.
Rationale:	Applications are considered potentially compromised by an attacker. Compromised applications can execute arbitrary logic and call arbitrary interfaces of the `Adaptive Platform Foundation`. Thus, access control cannot be enforced within the application but only in the trusted `Adaptive Platform Foundation`.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00514)*

[RS_SEC_03012] For each service interface an access control policy shall be specifiable ⌈

Type:	draft
Description:	For each service interface an access control policy shall be specifiable which describes what is needed to use that interface.
Rationale:	Crucial information for controlling another application (e.g. ADAS) could be provided via a service interface. Only an authorized Adaptive Application should be allowed to provide such a Service Interface.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00514)*

[RS_SEC_03013] The manifest shall provide a way to state a capability for each resource ⌈

Type:	draft
Description:	The manifest shall provide a way to state a capability for each resource. The `Adaptive Platform Foundation` shall deny access to the resource, if the application that wants to use the resource does not have the capability of the resource.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00514)*

[RS_SEC_03014] Unique Adaptive Application ID ⌈

Type:	draft

Description:	An Adaptive Application ID shall be unique regarding the local machine.
Rationale:	Adaptive Applications shall be linked to and held responsible for their actions.
Dependencies:	–
Use Case:	The IAM framework uses the application ID of Adaptive Applications to verify requests and grant access to certain resources based on the defined polices.
Supporting Material:	–

⌋(*RS_Main_00510*, *RS_Main_00514*)

[RS_SEC_03015] Unique Application Instance ID ⌈

Type:	draft
Description:	An Application Instance ID shall be unique regarding the local machine.
Rationale:	Applications shall be linked to and held responsible for their actions.
Dependencies:	–
Use Case:	The IAM framework uses the application ID to verify requests and grant access to certain resources based on the defined polices.
Supporting Material:	–

⌋(*RS_Main_00510*, *RS_Main_00514*)

[RS_SEC_03016] Termination of an application ID ⌈

Type:	draft
Description:	AUTOSAR shall support means to terminate application IDs that are no longer required.
Rationale:	Avoidance of unintended reuse of the access rights.
Dependencies:	–
Use Case:	Upon notice of a malicious application the corresponding application IDs has to be terminated. This also applies to any application that has been updated with a new application ID. Access rights of an application shall be removed upon termination (Stopping the application, preventing the application from further communication and preventing restart) of the application ID to prevent any unintended reuse of the access rights.
Supporting Material:	–

⌋(*RS_Main_00510*, *RS_Main_00514*)

[RS_SEC_03017] Set of capabilities shall be provided in the corresponding manifest ⌈

Type:	draft
Description:	The set of capabilities of an Adaptive Application shall be provided in the corresponding manifest.
Rationale:	To explicitly document capabilities it is required to ensure the correct behavior of an Adaptive Application. If an Adaptive Application is compromised, we need the manifest with the capabilities to actually enforce the restrictions implied by the capabilities. We cannot solely rely on the correct behavior of each Adaptive Application.
Dependencies:	–

Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_03018] Capabilities of an Adaptive Application shall be authentically linked to the manifest ⌈

Type:	draft
Description:	The set of capabilities of an Adaptive Application shall be authentically linked to the Adaptive Application in the corresponding application manifest.
Rationale:	An Adaptive Application is provided with a set of capabilities. It shall not be possible to extend or restrict this set. The Adaptive Application should always possess the same capabilities as originally intended.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

3.3 Secure Communication

[RS_SEC_04001] Secure communication shall be performed through secure channels ⌈

Type:	draft
Description:	Secure channels are established between communication nodes. Secure channels satisfy security requirements such as authenticity or confidentiality. Communication that is subject to specific security requirements is then routed through the respective secure channel.
Rationale:	Reduce resource consumption
Dependencies:	–
Use Case:	Nodes may host several services that could be multiplexed over a single secure channel.
Supporting Material:	–

⌋(*RS_Main_00503*, *RS_Main_00140*, *RS_Main_00514*, *RS_Main_00510*, *RS_Main_00280*, *RS_Main_00200*)

[RS_SEC_04002] Secure channels shall be defined ⌈

Type:	draft
Description:	Secure channels are established between communication nodes. Secure channels satisfy security requirements such as authenticity or confidentiality. Secure channels can be realized using various technologies. Therefore the kind and the parameters of a specific secure channel shall be configurable.
Rationale:	Enable modeling system security
Dependencies:	–

Use Case:	Nodes may host several services having different protection requirements thus requiring appropriate secure communication channels.
Supporting Material:	–

⌋(*RS_Main_00503*, *RS_Main_01005*, *RS_Main_00160*, *RS_Main_00150*, *RS_Main_00514*, *RS_Main_00280*)

[RS_SEC_04003] The assignment of communication to secure channels shall be defined ⌈

Type:	draft
Description:	Secure channels are established between communication nodes. Communication that is subject to specific security requirements is then routed through the respective secure channel.
Rationale:	Increase flexibility of Adaptive applications
Dependencies:	–
Use Case:	Specific communication shall be assigned to the appropriate secure channel.
Supporting Material:	–

⌋(*RS_Main_00503*, *RS_Main_01005*, *RS_Main_00106*, *RS_Main_00150*)

[RS_SEC_04004] Using secure channels shall be transparent on the communication API ⌈

Type:	draft
Description:	Communication through configured secure channels shall be facilitated transparently by the communication stack. Communicating through a secured channel shall make no difference for the application.
Rationale:	Reduce maintenance effort of Adaptive application code
Dependencies:	–
Use Case:	Application code may be reused to transfer data with different security requirements.
Supporting Material:	–

⌋(*RS_Main_00503*, *RS_Main_00150*, *RS_Main_00060*)

3.4 Protected Runtime Environment

3.4.1 Inter-Process Separation

Each two applications shall be running isolated. There shall be a spatial, time, and resource separation between the two applications.

[RS_SEC_05001] Individual virtual memory address space ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application uses only its individual virtual memory address space.

Rationale:	A shared virtual memory address space could lead to information leaking.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05002] Memory of other processes ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that a process does not access the memory of any other process. This includes access to information stored in memory freed by another process.
Rationale:	Access to memory of other processes can be exploited, even if it is freed by the other process.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05003] Access via communication management ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that communication between two applications is realized by Inter-Process Communication (IPC) mechanisms via Communication Management.
Rationale:	IPC is maximally secure only using the Communication Management.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05004] Individual persistency area ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application uses only its individual persistency area. The application shall not be allowed to access the persistency area of any other application.
Rationale:	Access to other applications' persistency area could lead to information leaks.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05005] Individual process tree ⌈

Type:	draft

Description:	The Autosar Platform shall ensure that an application uses its individual process tree. It shall not be aware of any process belonging to another application.
Rationale:	Otherwise processes gain information from each other.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05006] Maximum quota of memory allocation ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application is restricted to a predefined maximum quota of memory allocation and processor time.
Rationale:	This restriction is meant to ensure that other application also have enough resources available.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

3.4.2 Process-System Separation

[RS_SEC_05019] Access to Adaptive AUTOSAR Foundation and Services ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that access from an application to the Adaptive AUTOSAR Foundation and Services is restricted to a minimum.
Rationale:	Direct access to the foundation can be abused or misused.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05008] No physical memory address ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application does not access any physical memory address.
Rationale:	Direct access to a physical memory address can be abused or misused.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05009] Process access operating system functionality ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that either an application does not access operating system functionality via a system call at all, or the system call has to be authorized.
Rationale:	An application with direct access to the OS can cause significant damage, therefore the limitation / authorization.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05010] Inter-Process Communication ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application only uses Inter-Process Communication (IPC) to access the Adaptive AUTOSAR Foundation and Services.
Rationale:	The exclusive use of IPC is meant to ensure maximum security rsp. regulation.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

[RS_SEC_05011] Process access to the Adaptive AUTOSAR ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that process access to the Adaptive AUTOSAR Foundation and Services is authorized.
Rationale:	Direct access to AUTOSAR Foundation and Services can be abused.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

3.4.3 Protection against Memory Corruption Attacks

[RS_SEC_05012] The Adaptive Platform shall ensure that state-of-the-art protection mechanisms against memory corruption attacks are implemented. ⌈

Type:	draft

Description:	The Autosar Platform shall support the implementation of state-of-the-art protection mechanisms against memory corruption attacks.
Rationale:	Memory errors caused by error prone, unmanaged languages such as C or C++, are utilized to enforce memory corruption which is the root cause of nearly all vulnerabilities in todays software. If the vulnerabilities are exploited by a threat triggered by an attacker the possible impact is for example arbitrary code execution, privilege escalation, or exfiltration of sensitive information. A guideline for implementation is available as an Annex in this document. Note: The guideline is for reference only.
Dependencies:	Hardware, Operating System, Compiler
Use Case:	–
Supporting Material:	EXP_FCDesignSecurityManagement

⌋(*RS_Main_00514*)

3.4.4 Design Rules

[RS_SEC_05018] Privileges and permissions ⌈

Type:	draft
Description:	The Autosar Platform shall ensure that an application is only granted the privileges and permissions mandatory for fulfilling its intended purpose.
Rationale:	More privileges and permissions than required can lead to a higher impact in case of a compromised application.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋(*RS_Main_00514*)

4 Requirements Tracing

The following table references the features specified in [2] and links to the fulfillments of these.

Feature	Description	Satisfied by
[RS_Main_00060]	AUTOSAR shall provide a standardized software interface for communication between Applications	[RS_SEC_03010] [RS_SEC_04004]
[RS_Main_00106]	AUTOSAR shall provide the possibility to extend the software with new SWCs without recompiling the platform foundation	[RS_SEC_04003]
[RS_Main_00140]	AUTOSAR shall provide network independent communication mechanisms for applications	[RS_SEC_04001]
[RS_Main_00150]	AUTOSAR shall support the deployment and reallocation of AUTOSAR Application Software	[RS_SEC_04002] [RS_SEC_04003] [RS_SEC_04004]
[RS_Main_00160]	AUTOSAR shall provide means to describe interfaces of the entire system.	[RS_SEC_04002]
[RS_Main_00200]	AUTOSAR specifications shall allow resource efficient implementations	[RS_SEC_04001]
[RS_Main_00280]	AUTOSAR shall provide standardized communication interfaces for the onboard data exchange between ECUs with different software platforms	[RS_SEC_04001] [RS_SEC_04002]
[RS_Main_00330]	AUTOSAR shall support the principle of information hiding	[RS_SEC_03009]
[RS_Main_00503]	AUTOSAR shall provide a Software Platform that supports adaptation of communication topology after production	[RS_SEC_04001] [RS_SEC_04002] [RS_SEC_04003] [RS_SEC_04004]
[RS_Main_00510]	AUTOSAR shall support secure onboard communication	[RS_SEC_03014] [RS_SEC_03015] [RS_SEC_03016] [RS_SEC_04001]

[RS_Main_00514]	AUTOSAR shall support the development of secure systems	[RS_SEC_03001] [RS_SEC_03002] [RS_SEC_03003] [RS_SEC_03004] [RS_SEC_03005] [RS_SEC_03006] [RS_SEC_03007] [RS_SEC_03008] [RS_SEC_03009] [RS_SEC_03010] [RS_SEC_03011] [RS_SEC_03012] [RS_SEC_03013] [RS_SEC_03014] [RS_SEC_03015] [RS_SEC_03016] [RS_SEC_03017] [RS_SEC_03018] [RS_SEC_04001] [RS_SEC_04002] [RS_SEC_05001] [RS_SEC_05002] [RS_SEC_05003] [RS_SEC_05004] [RS_SEC_05005] [RS_SEC_05006] [RS_SEC_05008] [RS_SEC_05009] [RS_SEC_05010] [RS_SEC_05011] [RS_SEC_05012] [RS_SEC_05018] [RS_SEC_05019]
[RS_Main_01005]	AUTOSAR shall establish communication paths dynamically	[RS_SEC_04002] [RS_SEC_04003]

5 References

[1] Glossary
AUTOSAR_TR_Glossary

[2] Main Requirements
AUTOSAR_RS_Main

[3] Explanation of Adaptive Platform Design
AUTOSAR_EXP_PlatformDesign

[4] SoK: Eternal War in Memory

[5] Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR_RS_CPP14Guidelines

[6] The SPARC Architectural Manual, Version 8
http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz

[7] OpenBSD-3.3 announcement, public release of WX
http://www.openbsd.org/33.html

[8] Smashing The Stack For Fun And Profit
http://phrack.org/issues/49/14.html

[9] The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

[10] Jump-oriented Programming: A New Class of Code-reuse Attack

[11] On the Expressiveness of Return-into-libc Attacks

[12] Code-Pointer Integrity

[13] ARM Pointer Authentication
https://lwn.net/Articles/718888/

[14] PaX ASLR (Address Space Layout Randomization)

[15] Control-flow Integrity

[16] AMD64 Architecture Programmer's Manual Volume 2: System Programming
http://support.amd.com/TechDocs/24593.pdf

[17] PowerPC Architecture Book, Version 2.02
https://www.ibm.com/developerworks/systems/library/es-archguide-v2.html

[18] PowerPC Operating Environment Architecture Book III
http://public.dhe.ibm.com/software/dw/library/es-ppcbook3.zip

[19] Linux Kernel, Summary of changes from v2.6.7 to v2.6.8
https://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.8

[20] PAX
https://pax.grsecurity.net/docs/pax.txt

[21] CPI LLVM on github
https://github.com/cpi-llvm

[22] Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors

[23] Drammer: Deterministic Rowhammer Attacks on Mobile Platforms

[24] ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks

[25] A seccomp overview
https://lwn.net/Articles/656307/

[26] Frequently Asked Questions for FreeBSD 10.X and 11.X

[27] pledge(2)
https://man.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2

A Protected Runtime Environment

A.1 Introduction

Vulnerabilities in software programs lead to unauthorized system manipulation and access when they are exploited by runtime attacks. Unauthorized system manipulations are, for instance, arbitrary code execution, privilege escalation, or persistent manipulation of storage. The cause of the vulnerabilities are mostly programming mistakes and design flaws. Although design rules are known during the development process or quality assurance processes like static code analysis or fuzzing are performed, vulnerabilities exist statistically in nearly all projects. These measures can being specified only qualitatively by the AUTOSAR Adaptive Platform specification. However, there are technical countermeasures on the operating system level to harden a system against such attacks. Note that the term harden includes that there is still no guaranteed system security but the effort for a successful attack can be raised to a higher level.

The hardening measures are combined as the term *Protected Runtime Environment* (PRE) in the context of AUTOSAR Adaptive Platform. A PRE includes the most important, basic protection mechanisms for a complex software environment. Without it, any other security mechanism will be circumventable, as any compromised process or service will be able to compromise any other running process on a system. The goal of a protected runtime environment is to protect the integrity of a process' control-flow during runtime and to limit the impact of a successful attack. So, two strategies of hardening are considered for AUTOSAR Adaptive Platform: proactive protection that should mitigate the exploitation of vulnerabilities and "post-mortem" measures that isolates an untrusted process.

The present document is a guidance for integrators and implementers who are going to develop an automotive system that is compliant to the AUTOSAR Adaptive Platform specifications. It contains recommendations and hints to fulfill the desired security requirements listed in this document. The actual implementation of a specific measure is yet to be defined and may depend on the concrete system at hand, or may be a combination of multiple measures.

Section A.2 introduces exploit mitigation approaches and presents related integration options. Afterwards, Section A.3 discusses the isolation aspects that limits the action scope of a compromised or untrusted process. In each chapter the general attack and mitigating techniques are detailed and existing countermeasure implementations are presented. Further, technical requirements for the integration are highlighted.

A.2 Protection against Memory Corruption Attacks

Unmanaged languages, such as C or C++, enable programmers to implement their code with a high degree of freedom to resource management. As such, code can be optimized for runtime performance or memory consumption and access to low level functions of the operating system is possible. However, programmers are fully respon-

sible for bounds checking and memory management since C/C++ is memory-unsafe. In practice, this often leads to the violation of the *Memory Safety* policy or memory errors, as the manual management of memory is very error prone.

Memory errors in turn can be utilized to cause memory corruption, the root cause of nearly all vulnerabilities in software components. If the vulnerabilities are exploited by an attacker the possible impact is, for example, arbitrary code execution, privilege escalation, or the leakage of sensitive information.

The language of choice in AUTOSAR AP is C++14 as proposed in platform design explanatory document [3, Section 2.3.1]. As such, the designated programming language for Adaptive Applications (AA), AUTOSAR Runtime for Adaptive Applications (ARA), as well as Functional Clusters on the Adaptive Platform Foundation or the Adaptive Platform Services is unmanaged, resulting in a large attack surface. One goal of a PRE is the minimization of this attack surface.

This chapter is structured as follows. Section A.2.1 gives an overview of the potential types of memory corruption vulnerabilities. The typical pitfalls during coding are detailed in section A.2.2. In section A.2.3 an explanation of the basic technical reasons of memory corruption attacks and the corresponding state-of-the-art protection techniques on various attack stages is given. Further, possible practical solutions to implement and integrate the presented protection techniques in terms of the AUTOSAR AP are presented in Section A.2.4.

A.2.1 Overview

The exploitation of vulnerabilities and its mitigation is a complex topic. Computer security researchers continuously develop new attacks and corresponding defenses. A general model for memory corruption attacks and the corresponding protecting techniques is described in [4]. The model (cf. Figure A.1) summarizes the general causes of vulnerabilities, the way how to exploit them according to the targeted impact, as well as mitigation policies on the individual attack stages for four types of attacks: *Code corruption attack*, *Control-flow hijack attack*, *Data-only attack*, and *Information leak*. On each attack stage they define several policies that must hold to prevent a successful attack.

The first two stages are common for all attack types and describe the root cause of vulnerabilities. In the first stage a memory corruption manipulates a pointer. When this *invalid* pointer is then dereferenced, a corruption is triggered. A pointer is invalid if it is an *out-of-bounds* pointer, i.e. pointing out of the bounds of a previously allocated memory area, or if it becomes a *dangling pointer*, i.e. pointing to a deleted object. Commonly known out-of-bounds vulnerabilities are for example: *buffer over-/underflow*, *format string bug*, and *indexing bugs* like *integer overflow*, *truncation* or *signedness bug*, or *incorrect pointer casting*. Typical dangling pointer vulnerabilities are: *use-after-free* or *double-free*. A collection of C and C++ related issues can be found, for instance, in the *List of Software Weakness Types* of the *Common Weakness Enumeration (CWE)* from

MITRE[1]. The exploration of memory errors is the first step of an attack. Subsequently a pointer is dereferenced to read, write or free memory.

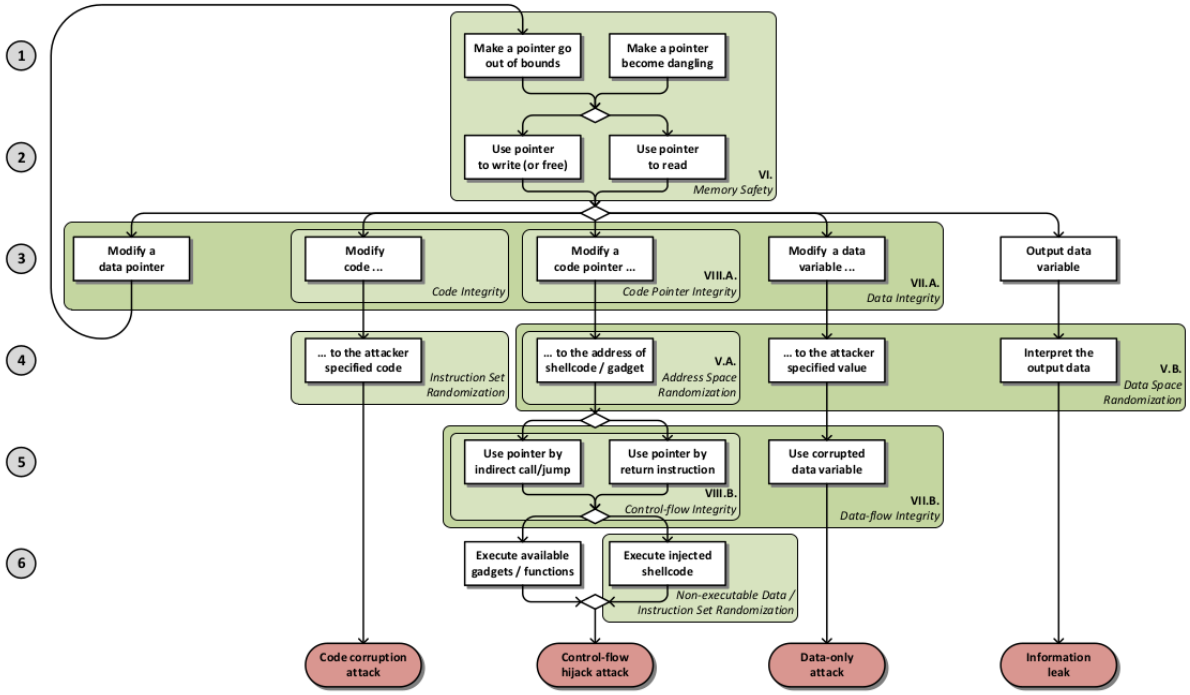

Figure A.1: Attack model from [4] demonstrating four attack types, policies mitigating the attacks in different attack stages

A.2.2 Secure Coding

A first measure to counter vulnerabilities at their root is to avoid mistakes and errors in the first place. To reach this goal programmers have to take care of many pitfalls during the development process. A simple example is the usage of unsafe functions from the standard C library like `strcpy()`. It copies a null character terminated character string to a buffer until the null character is reached. If the allocated destination buffer is not large enough, the function still copies characters behind the end of the buffer and thus overwrites other data. This is one of many pitfalls commonly known as a *buffer overflow* and can be used by an attacker, for example, to overwrite the stored return pointer if the buffer is allocated on the stack. For the given example a programmer should use safer variants instead. To that end, many standard C library functions have been supplemented with versions including a bounds check, for `strcpy()` this is `strncpy()`. Therewith the length of the input is limited and the buffer, if it is allocated properly, does not get overflowed. While there are supposedly more safe functions, such as `strncpy()`, they come with their own quirks and flaws. But also more complex, context related issues must be considered.

[1] `http://cwe.mitre.org/data/definitions/659.html`

In practice programmers should have coding guidelines at hand like the *MISRA* for safety-related systems. Unfortunately the AUTOSAR Coding Guideline does not cover a rule set for security related issues [5]. But there are third party guidelines which deals with these issues, like the *SEI CERT C++ Coding Standard*[2].

Since these guides are very comprehensive only few programmers will follow them in practice due to time reasons. However, there are some tools that can support the check against some rules in a static code analysis, e.g. *Flawfinder*[3], *RATS*[4], or *CodeSonar*[5]. But nevertheless the application of tools does not guarantee vulnerability-free code finally since there are still several runtime conditions and contexts the tools could not investigate or the vulnerabilities are not fixed properly.

A.2.3 Attacks and Countermeasures

For a second line of defense it is assumed that vulnerabilities are present and that never all vulnerabilities are avoided by programmers. Therefore the requirement for enhanced countermeasures is the independence from written code. The goal is to mitigate that vulnerabilities get exploited.

A.2.3.1 Code Corruption Attack

A *Code Corruption Attack* intends to manipulate the executable instructions in the text segment in the virtual memory space and to breach the *Code Integrity* policy (cf. Figure A.1). The countermeasure is to set the memory pages containing code to read-only or W $\oplus$ X (write xor execute), respectively. It has to be implemented on both system levels, i.e. the processor as well as the operating system. The MMU of the CPU has to provide fine-grained memory permission layout (e.g. NX-bit [6, p.248]) or the operating system should emulate this. Further, the operating system level has to support the underlying permission layout, e.g. like W $\oplus$ X [7] or Data Execution Prevention (DEP) . But care has to be taken if self-modifying code and Just-In-Time (JIT) compilation is used, as the generated code must first be written to writeable pages, which are then set to be executable.

A.2.3.2 Control-flow Hijack Attack

A *Control-flow Hijack Attack* starts with the exploitation of a memory corruption to modify a code pointer so that it points to an attacker defined address and the *Code Pointer*

[2] https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

[3] https://www.dwheeler.com/flawfinder/

[4] https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml

[5] https://www.grammatech.com/products/codesonar

Integrity policy is harmed (cf. Figure A.1). This pointer is then used by a indirect control flow transfer in the original code. Therewith the control-flow is diverted from the original and so its *Control-flow Integrity* is violated. The last step is the execution of the exploit payload. The literature distinguishes between two approaches: *code-injection attacks* and *code-reuse attacks*. While code-injection attacks [8] are based on injecting arbitrary and custom instructions (a.k.a. *shellcode*) into the memory as exploit payload, code-reuse attacks, such as *return-oriented program* (ROP) [9], *jump-oriented programming* (JOP) [10], and *return-to-libc* [11], utilize existing code in the process memory to construct so called *gadgets*, which enable the targeted malicious functionality.

All in all, a control-flow hijack attack will be successful if the integrity of a code pointer and of the control-flow are broken. Further, the value of the target address of the malicious functionality must be known and in the case of code-injection, the memory pages holding the injected code must be executable.

The goal of *Code Pointer Integrity* is satisfied if all dereferences that either dereference or access sensitive pointers, such as direct and references to code pointers, are not modified (cf. [12]). There are a few recent feasible approaches which detect the alteration of code pointers at this early stage. The *Code Pointer Integrity* (CPI) [12] mechanism provides full memory safety but just for direct and references to code pointers. According to the authors, this approach protects against all control-flow hijack attacks. CPI is a combination of static code analysis to identify all sensitive pointers, rewrite the program to protect identified pointers in a safe memory region, called *safe-stack*, and instruction level isolation that controls the access to the safe region. These mechanisms require both compiler and runtime support and comes with an overhead of ca. 8% on runtime. An additional requirement is Code Integrity. *Code Pointer Separation* (CPS) [12] is a relaxation of CPI. Among others, CPS limits the set of protected pointers to code pointer (no indirections) to lower the overhead to ca. 2%. It still has strong detection guarantees. A further approach to detect modification on code pointers is *Pointer Authentication* [13]. This approach uses cryptographic functions to authenticate and verify pointers before dereferencing. Pointers are replaced by a generated signature and cannot be dereferenced directly. This requires compiler and instruction set support. To reduce overhead, hardware acceleration for the cryptographic primitives is required.

With *Stack Smashing Protection* (SSP) stack-based buffer overflows, which overwrite the saved return address, can be detected. Therefore, a pseudo-random value, also known as *Stack Canary* or *Stack Cookie*, is inserted on the stack before the saved base pointer and the saved return address as part of the function prologue. When the function returns and the function epilogue is performed, the value is compared to a protected copy. If the value is overwritten by a buffer overflow which targets the return address, actually, the program aborts because the values do not match anymore.

As mentioned before, code-injection attacks require executable memory pages for the injected instructions. With principle of $W \oplus X$, also called *Data Execution Prevention*(DEP), a memory page is either flagged as writable or executable, but not both. This prevents that instructions overwrite data memory such as stack or heap and ex-

ecute it afterwards. The approach requires a fine-grained page permissions support either by the MMU of the CPU and the so called *NX-bit* (No-execute bit) or emulated in Software as described in Section A.2.3.1. Moreover, the employed operating system must support it.

Code-reuse attacks are not affected by the W $\oplus$ X mechanism since existing, executable marked, memory regions are utilized and no additional code must be injected to the memory. To mitigate such kind of attacks currently deployed countermeasures are implemented on previous attack stages. On the fourth attack stage it is stated that the target address of the malicious functionality must be known. In general, the attacker knows or can just estimate an address in the virtual address space since it is static for a binary after compilation. A countermeasure in practice is the obfuscation of the address space layout by *Address Space Layout Randomization* (ASLR) [14]. Therewith the locations of various memory segments get randomized which makes it harder to predict the correct target addresses. ASLR requires high entropy to prevent brute-force de-randomization attacks and depends on the prevention of unintended Information Leak (Section A.2.3.4) that are used by dynamically constructed exploit payloads. To guarantee high entropy ASLR should be implemented on 64-bit architectures (or above). Additionally, every memory area must be randomized, including stack, heap, main code segment, and libraries.

In addition to ASLR the policy Control-flow Integrity intends to detect a diversion of the original control-flow. Established techniques are: *Shadow Stack* and Control-flow integrity (Abadi) (CFI) [15]. The idea of shadow stack is to push the saved return address to a separate shadow stack so that it is compared upon a function return. In addition, CFI also protects indirect calls and jumps as well. The original CFI creates a static control-flow graph by determining valid targets statically and give them a unique identity (ID). Afterwards, calls and returns are instrumented to compare the target address to the ID before jumping there. It is required to protect valid targets from overwrite by W $\oplus$ X.

A.2.3.3 Data-only Attack

A memory corruption can also be exploited to modify security critical data that is not related to control-flow data. For instance, exploiting a buffer overflow to alter a conditional construct can lead to unintended program behavior. Therewith the policy *Data Integrity* is violated. Techniques such as *Data Space Randomization* and *Write Integrity Testing* (WIT) makes it harder to perform such kinds of attacks but they are not established in practice yet.

A.2.3.4 Information Leak

Memory corruption attacks are also used to leak memory contents. Therewith probabilistic countermeasure like ASLR can be circumvented by the knowledge of randomly

generated data. As for the data-only attack *Data Space Randomization* might help to mitigate information leakage.

A.2.4 Existing Solutions

In this section current state-of-the-art solutions of implemented countermeasures mentioned in the sections before are presented. For each approach the system level (compiler, operating system, or hardware) at which an approach is enforced is called and which technical and security requirements are expected.

A.2.4.1 W $\oplus$ X, Write xor Execute, Data Execution Prevention (DEP)

System Level: Hardware, Operating System

The idea of this approach is to flag memory pages either writable or executable, but not both at the same time. Therewith code-injected attacks are mitigated. At the lowest system level a mechanism for fine-grained memory page permissions is required. Further the operating system must support the hardware mechanism or even emulate a memory page permission mechanism.

Architecture	Instruction Set	Enforcement
x86	AMD64 [16, p. 56] Intel64	No-eXecution bit (NX-bit), Page Table eXecute Disable (XD-bit), Page Table
ARM	ARMv6 ARMv8-A	execute never bit (XN-bit), Page Table PEN privileged execute never, PAN privileged access never
SPARC	Oracle SPARC Architecture 2011	Translation Storage Buffer (optional)
PowerPC	IBM PowerISA [17] [18, p. 33]	Segment Lookaside Buffer (SLB)

Examples of Hardware Support for Execution Prevention

Family	Name	Implementation
Linux	Linux kernel	Linux kernel mainline $\geq$ 2.6.8 [19]
	PaX	Patch for the Linux kernel, uses hardware support or emulates memory page permission [20]
	Exec Shield	Patch for the Linux kernel, part of Fedora Core 1 through 6[6] and Red Hat Enterprise Linux $\geq$ 3[7][8], uses hardware support or emulates memory page

[6] https://archives.fedoraproject.org/pub/archive/fedora/linux/core/1/x86_64/os/RELEASE-NOTES.html

[7] http://people.redhat.com/mingo/exec-shield/

[8] https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

	grsecurity	Patch for the Linux kernel, uses hardware support or emulates memory page permission[9][10]
	Android	Android $\geq$ 2.3[11]
Unix	OpenBSD	OpenBSD $\geq$ 3.3[12]
	NetBSD	NetBSD $\geq$ 2.0[13]
	FreeBSD	FreeBSD $\geq$ 5.3

Examples of Operating System Support for Execution Prevention

A.2.4.2 Stack Smashing Protection (SSP)

System Level: Compiler

Stack Smashing Protection (SSP) mechanisms place pseudo-random values (*Stack Canaries* or *Stack Cookie*) on the stack before the saved base pointer and the saved return address as part of the function prologue and compare the value again before a function returns. SSP is enforced at compile-time. Due to performance reasons, the compiler has to decide which function has to be protected. If the compiler implementation makes the wrong decision and the concerned function is vulnerable, SSP fails. Further, information leaks enable an attacker to read the pseudo-random value and integrate it to her exploit so that the buffer is overwritten with the correct value.

Compiler	Option	Description
GNU Compiler Collection (GCC)[14]	`-fstack-protector`	Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.
	`-fstack-protector-all`	Like -fstack-protector except that all functions are protected.
	`-fstack-protector-strong`	Like -fstack-protector but includes additional functions to be protected – those that have local array definitions, or have references to local frame addresses.

[9] `https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Enforce_non-executable_kernel_pages`
[10] `https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Paging_based_non-executable_pages`
[11] `https://source.android.com/security/#memory-management-security-enhancements`
[12] `http://www.openbsd.org/33.html`
[13] `http://www.netbsd.org/docs/kernel/non-exec.html`
[14] `https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Instrumentation-Options.html#Instrumentation-Options`

	`-fstack-protector-explicit`	Like -fstack-protector but only protects those functions which have the stack_protect attribute.
Clang[15]	`-fstack-protector-all`	Force the usage of stack protectors for all functions.
	`-fstack-protector-strong`	Use a strong heuristic to apply stack protectors to functions.
	`-fstack-protector`	Enable stack protectors for functions potentially vulnerable to stack smashing.
Intel C++ Compiler[16]	`-fstack-security-check`	This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.
Keil ARM C/C++ Compiler[17]	`-protect_stack`	Use `-protect_stack` to enable the stack protection feature. Use `-no_protect_stack` to explicitly disable this feature. If both options are specified, the last option specified takes effect.
	`-protect_stack_all`	The `-protect_stack_all` option adds this protection to all functions regardless of their vulnerability.

Examples of Stack Smashing Protection Compiler Support

A.2.4.3 Address Space Layout Randomization (ASLR)

System Level: Compiler, Operating System

Address Space Layout Randomization (ASLR) obfuscates the address space layout of a process. Therewith the locations of various memory segments get randomized which makes it harder to predict the correct target address which is needed to perform a code-reuse attack. ASLR requires high entropy to prevent brute-force de-randomization attacks and depends on the prevention of unintended Information Leak (Section A.2.3.4) that are used by dynamically constructed exploit payloads. To guarantee high entropy ASLR should be implemented on 64-bit architectures (or above). Additionally, ASLR requires position-independent executables (PIE) which implements a random base address for the main executable binary.

ASLR is enforced by the operating system primarily but requires position-independent executables for binaries and position independent code for shared libraries generated by the complier.

Family	Name	Implementation

[15] `https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fstack-protector`
[16] `https://software.intel.com/en-us/node/523162`
[17] `http://www.keil.com/support/man/docs/armcc/armcc_chr1359124940593.htm`

Linux	Linux kernel	Linux kernel mainline $\geq$ 3.14[18]. ASLR for user-space programs and Kernel Address Space Layout Randomization (KASLR)[19] for the kernel itself.
	PaX	Patch for the Linux kernel[20]
	Exec Shield	Patch for the Linux kernel[21]
	grsecurity	Patch for Linux kernel[22]
	Android	Android $\geq$ 4.1[23]
BSD	OpenBSD	OpenBSD $\geq$ 4.4[24]
	NetBSD	NetBSD $\geq$ 5.0[25]
	FreeBSD	FreeBSD as patch[26]

Examples of ASLR Operating System Support

Compiler	Option	Description
GNU Compiler Collection (GCC)[27][28]	`-fpie, -fPIE`	These options are similar to `-fpic` and `-fPIC`, but generated position independent code can be only linked into executables. Usually these options are used when `-pie` GCC option is used during linking. This is especially difficult to plumb into packaging in a safe way, since it requires the executable be built with `-fPIE` for any `.o` files that are linked at the end with `-pie`. There is some amount of performance loss, but only due to the `-fPIE`, which is already true for all the linked libraries (via their `-fPIC`).
	-fPIC	If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on AArch64, m68k, PowerPC and SPARC. Position-independent code requires special support, and therefore works only on certain machines.
Clang[29]	`-fpie, -fPIE`	See GCC.

[18] https://lwn.net/Articles/569635/
[19] http://selinuxproject.org/~jmorris/lss2013_slides/cook_kaslr.pdf
[20] https://pax.grsecurity.net/docs/aslr.txt
[21] http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf
[22] https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Restrict_mprotect.28.29
[23] https://source.android.com/security/enhancements/enhancements41
[24] https://www.openbsd.org/plus44.html
[25] https://netbsd.org/releases/formal-5/NetBSD-5.0.html
[26] https://hardenedbsd.org/content/freebsd-and-hardenedbsd-feature-comparisons
[27] https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Code-Gen-Options.html#Code-Gen-Options
[28] https://wiki.debian.org/Hardening#gcc_-pie_-fPIE
[29] https://clang.llvm.org/docs/ControlFlowIntegrity.html

	`-fPIC`	See GCC.
Intel C++ Compiler[30][31]	`-pie`	Determines whether the compiler generates position-independent code that will be linked into an executable.
	`-fpic`	Determines whether the compiler generates position-independent code.
Keil ARM C/C++ Compiler[32]	`-bare_metal_pie`	(Bare-metal PIE support is deprecated. There is support for `-fropi` and `-frwpi` in armclang. You can use these options to create bare-metal position independent executables.) A bare-metal Position Independent Executable (PIE) is an executable that does not need to be executed at a specific address but can be executed at any suitably aligned address.
	`-fropi, -fno-ropi`	Enables or disables the generation of Read-Only Position-Independent (ROPI) code.
	`-frwpi, -fno-rwpi`	Enables or disables the generation of Read/Write Position-Independent (RWPI) code.

Examples of ASLR Compiler Support

A.2.4.4 Control-flow Integrity (CFI)

System Level: Compiler

Control-flow Integrity (CFI) is a current research topic. However, Clang includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow. These schemes have been optimized for performance, allowing developers to enable them in release builds. The CFI implementation in Clang has a performance overhead of ca. 1% and a binary size overhead of ca. 15% (only forward-edges)[33].

Compiler	Option	Description

[30] `https://software.intel.com/en-us/node/523278`
[31] `https://software.intel.com/en-us/node/523158`
[32] `http://www.keil.com/support/man/docs/armclang_dev/armclang_dev_chr1405439371691.htm`
[33] `https://clang.llvm.org/docs/ControlFlowIntegrity.html`

GNU Compiler Collection (GCC)[34]	`-fvtable-verify=[std|preinit|none]`	This option is only available when compiling C++ code. It turns on (or off, if using `-fvtable-verify=none`) the security feature that verifies at run time, for every virtual call, that the vtable pointer through which the call is made is valid for the type of the object, and has not been corrupted or overwritten. If an invalid vtable pointer is detected at run time, an error is reported and execution of the program is immediately halted. This option causes run-time data structures to be built at program startup, which are used for verifying the vtable pointers. The options `std` and `preinit` control the timing of when these data structures are built. In both cases the data structures are built before execution reaches main. Using `-fvtable-verify=std` causes the data structures to be built after shared libraries have been loaded and initialized. `-fvtable-verify=preinit` causes them to be built before shared libraries have been loaded and initialized. If this option appears multiple times in the command line with different values specified, `none` takes highest priority over both `std` and `preinit`; `preinit` takes priority over `std`.
Clang[35]	`-fsanitize=cfi-cast-strict`	Enables strict cast checks.
	`-fsanitize=cfi-derived-cast`	Base-to-derived cast to the wrong dynamic type.
	`-fsanitize=cfi-unrelated-cast`	Cast from void* or another unrelated type to the wrong dynamic type.
	`-fsanitize=cfi-nvcall`	Non-virtual call via an object whose vptr is of the wrong dynamic type.

[34] `https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Code-Gen-Options.html#Code-Gen-Options`

[35] `https://clang.llvm.org/docs/ControlFlowIntegrity.html`

	`-fsanitize=cfi-vcall`	Virtual call via an object whose vptr is of the wrong dynamic type.
	`-fsanitize=cfi-icall`	Indirect call of a function with wrong dynamic type.
	`-fsanitize=cfi`	Enable all the schemes.

Examples of CFI Compiler Support

A.2.4.5 Code Pointer Integrity (CPI), Code Pointer Separation (CPS)

System Level: Compiler

Code-Pointer Integrity (CPI) is a property of C/C++ programs that guarantees absence of control-flow hijack attacks by requiring integrity of all direct and indirect pointers to code. Code-Pointer Separation (CPS) is a simplified version of CPI that provides strong protection against such attacks in practice. SafeStack is a component of CPI/CPS, which can be used independently and protects against stack-based control-flow hijacks.

CPI/CPS/SafeStack can be automatically enforced for C/C++ programs through compile-time instrumentation with low performance overheads of 8.5% / 1.9% / 0.05% correspondingly. The SafeStack enforcement mechanism is now part of the Clang compiler, while CPI and CPS are available as research prototypes. For the current status please see [21].

Compiler	Option	Description
Clang[36][37]	`-fsanitize=safe-stack`	SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.

Examples of CPI and CPS Compiler Support

[36] `http://dslab.epfl.ch/proj/cpi/`
[37] `https://clang.llvm.org/docs/SafeStack.html`

A.2.4.6 Pointer Authentication

System Level: Hardware, Compiler

Pointer Authentication uses cryptographic functions to authenticate and verify pointers before dereferencing [13]. Pointers are replaced by a generated signature and cannot be dereferenced directly. This requires compiler and instruction set support. ARM recently introduces the Pointer Authentication extensions in ARMv8.3-A specification[38]. The GCC introduces basic support for pointer authentication in version 7[39] for the *AArch64* target (ARMv8.3-A architecture). The overhead is negligible because of hardware acceleration for the cryptographic primitives[40].

Compiler	Option	Description
GNU Compiler Collection (GCC)[41]	-msign-return-address= scope	Select the function scope on which return address signing will be applied. Permissible values are `none`, which disables return address signing, `non-leaf`, which enables pointer signing for functions which are not leaf functions, and `all`, which enables pointer signing for all functions. The default value is `none`.

Examples of Pointer Authentication Compiler Support

A.3 Isolation

Isolating software components within a system is a common protection measure to protect other components from erroneous ones, either through unintentional programming errors, or intentional harm caused by an attacker taking over a corruptible component. While the protective measures detailed in section A.2 are a preventive measure, intended to impede an attacker from taking over a software component by exploiting programming errors, isolation intends to limit the influence an attacker might have to other software components *after* taking over a software component. As such, isolation is only an effective measure, if the architecture of the system is appropriately designed, dividing and isolating different functional aspects of the system accordingly. Note that this guide does not cover this architectural aspect of isolation, but the technical aspect, i.e. *how* isolation can be implemented.

The following sections describe two approaches to isolation: the isolation of multiple software components between each other, and the isolation of software components and the operating system itself. These two approaches are orthogonal – i.e. horizontal

[38] `https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions`
[39] `https://gcc.gnu.org/gcc-7/changes.html`
[40] `https://lwn.net/Articles/719270/`
[41] `https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/AArch64-Options.html#AArch64-Options`

isolation between applications and vertical isolation between applications and the OS – and can be combined accordingly.

A.3.1 Horizontal Isolation

A.3.1.1 Virtual Memory

The oldest and most prevalent isolation mechanism is the concept of virtual memory, i.e. presenting each running software component of a system with its own virtual address space, which is mapped to the available physical memory by the operating system. The origins date back to the 1950s, with the original intention of hiding the fragmentation of physical memory, but offers the possibility of isolating software components. As each component operates in a virtual address space, it cannot access the memory of other components (unless explicitly allowed by the operating system). This feature requires hardware support, e.g. by a Memory Management Unit (MMU), but this support is nearly ubiquitous in most computer systems except very small microcontrollers.

An extension of this concept is that of *virtual machines* (or *virtualization*), whereby multiple virtual machines (VM) are emulated by a *hypervisor*. Each VM may run a complete operating system, depending on the degree of virtualization even in an unmodified state. The hypervisor controls the access of each VM to the physical hardware components, or even emulate certain components such as network interfaces. Similar to virtual memory, the virtualization requires dedicated hardware support to achieve an appropriate level of performance and security.

Note that both approaches have limitations. The isolation provided by virtual memory or virtualization is only as strong as the operating system or hypervisor itself, as a malicious application might take control over the OS or hypervisor through a programming error (the measures described in section A.4 intend to minimize this attack surface). Similarly, a malicious application might use its access to other hardware components to circumvent the isolation, as some hardware components may have unrestricted access to the systems memory. “IOMMUs”, a technique which presents hardware components with a virtual address space, can be used to counter this. Lastly, an attacker might use the volatile properties of physical memory itself to circumvent the isolation. For example, in [22] the “rowhammer” attack is described, which is capable of flipping bits in memory locations usually inaccessible to an application. The attack uses prolonged reads to a memory location performed in quick succession, which in the case of “DRAM” memory will cause neighboring memory cells to change state. This effect has been shown to be capable of raising the privileges of an application in Linux and Android (cf. [23]) systems. System designers must consider using one or more mitigations to such attacks, for example, by using memory with error correction, or software mitigations as shown in [24].

A.3.2 OS-Level Virtualization

A more lightweight form of virtualization, often called operating-system-level virtualization or *containerization*, is available in modern OS. Prominent examples are “LXC” [42], which is built on top of the Linux Kernel Namespacing functionality, or the “Jail”[43] functionality provided by the FreeBSD operating system. These tools only virtualize certain resources of an operating system, for example, file system, the process tree, or the network stack. This way, the operating system creates a *container* with a tightly controlled access to system resources or other containers. In contrast to full virtualization, these containers cannot execute a different operating system, albeit a completely separate user-space instance can be run side-by-side.

OS Family	Solution	Description
UNIX	chroot	This is a system call available in many UNIX alike systems, which can be used to set up a an execution environment with a different root filesystem. As such, it only isolates the filesystem of the host from the container. If the container contains privileged processes, they can easily affect the system
Linux	Namespaces, LXC, Docker, systemd-nspawn	Many solutions building upon the Linux Namespacing functionality exist, many allowing the setup of isolated containers. Resources of other containers or the host are not visible to the processes running inside these containers, unless assigned. This way, even privileged processes inside a container are not capable of affecting the rest of the system.
FreeBSD	Jails	A solution similar to chroot, except for improved security. For example, these containers (or called “jails” in this context) offer an isolated network, as well as restricting the capabilities of privileged processes inside (e.g. privileged processes cannot affect the rest of the system).

Operating System Virtualization / Container Implementations

A.4 Vertical Isolation

Isolating the operating system from applications, often called *sandboxing* is an another important aspect of a protected runtime environment. The basic idea is to limit the

[42] https://linuxcontainers.org/
[43] https://www.freebsd.org/doc/handbook/jails.html

capabilities of a process, i.e. restricting what a process can do[44]. The classic way to do this is by dropping privileges as soon as they are not needed anymore, i.e. *priviledge revocation*. For example, the `ping` command requires root privileges on UNIX systems to create a raw network socket, but will drop its privileges to a regular user after creating it. Ideally, a software component should drop (or never be given) any privileges it does not require, or drop them as soon as they are no longer required. As with the example of `ping`, any software component must then be structured with a setup phase, in which all advanced privileges are used and subsequently dropped. The following shows a few examples of operating system functionalities, which allow an application to drop capabilities or privileges.

OS Family	Solution	Description
Linux	Seccomp Mode 1 (Strict) [25]	Processes on the Linux operating system can limit their set of allowed system calls in a very easy manner. The allowed subset is extremely strict, limiting the process to `read`, `write`, `exit` and `sigreturn`. Once activated, this Seccomp mode cannot be deactivated again.
Linux	Seccomp Mode 2 (Filter) [25]	A more recent version of the Seccomp mode, the seccomp filter mode, allows a much more fine-grained control. The concept is to allow a process to attach a small filter program, which will check each system call. This filter program must be a “Berkley Packet Filter” (BPF), a very restricted form of byte-code, which will be executed by the kernel before each system call. This filter can than examine each system call, for example, check which system call is made or what the parameters are set. The filter then returns a decision as to what the kernel should do. The system call can be allowed or blocked and if the call is blocked, several choices can be made as to how it is blocked. The filter may decide to immediately kill the process, to simply let the system call return an error, send a signal to the offending process or to notify a tracer attached to the program (such as a debugger). A notable property of these filters is, that they are inherited by the spawned child processes, which enables a setup of a filter before a potentially dangerous process is started.
FreeBSD	Securelevel [26, chapter 13]	This functionality limits the possibilities of all processes running on a system in incremental levels, which cannot be lowered once entered (until a reboot of the system).

[44] Not to be confused with “what a process does”, as the behaviour of a process is changed by an attack.

OpenBSD	Pledge [27]	This functionality can be used to limit a processes system calls. Certain system calls can be prohibited completely, others can be restricted in their functionality. For example, the `open` system call can be limited to only open a small set of system files, or the `mprotect` cannot set memory to be executable. Once the limitations are in place, the process cannot lift them again.

Sandoxing Mechanisms in Operating Systems