Document Title	Explanation of Security Overview
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	1077

Document Status	published
Part of AUTOSAR Standard	Foundation
Part of Standard Release	R23-11

Document Change History

Date	**Release**	**Changed by**	**Description**
2023-11-23	R23-11	AUTOSAR Release Management	• Updated chapter Secure Coding introducing RAII idiom.
2022-11-24	R22-11	AUTOSAR Release Management	• Initial release

Contents

1 Introduction

This explanatory document provides additional information regarding secure design for the AUTOSAR standards. This document is currently limited to the AUTOSAR `Adaptive Platform`. Support for the AUTOSAR `Classic Platform` may be added in a future release of this document.

1.1 Objectives

This document explains security features which could be utilized within the AUTOSAR `Adaptive Platform`. The motivation is to provide standardized and portable security for `Adaptive Applications` as well as the whole AUTOSAR `Adaptive Platform`.

1.2 Scope

This document shall be explanatory and help the security engineer to identify security related topics within `Adaptive Application`s and the AUTOSAR `Adaptive Platform`.

The content of this document will address the following topics:

- Protection against memory corruption attacks.
- Isolation of software components between each other.
- Isolation of the operating system from software components.
- Existing security solutions

2 Definition of terms and acronyms

2.1 Acronyms and abbreviations

All acronyms used are included in the AUTOSAR TR Glossary.

3 Related Documentation

3.1 Input documents & related standards and norms

[1] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[2] SoK:Eternal War in Memory

[3] C++ Core Guidelines
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

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

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

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

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

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

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

[10] Code-Pointer Integrity

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

[12] PaX ASLR (Address Space Layout Randomization)

[13] Control-flow Integrity

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

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

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

[17] 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

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

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

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

[21] Drammer:Deterministic Rowhammer Attacks on Mobile Platforms

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

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

[24] Frequently Asked Questions for FreeBSD 10.X and 11.X
https://www.freebsd.org/doc/en/books/faq/security.html

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

4 Security Overview

4.1 Protected Runtime Environment

4.1.1 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 and guidelines might be followed 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 be 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's 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 fulfil 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 4.1.2 introduces exploit mitigation approaches and presents related integration options. Afterwards, Section 4.1.7 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 prerequisites for the integration are highlighted.

4.1.2 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 manage resources. 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 responsible 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 the AUTOSAR `Adaptive Platform` is C++14 as proposed in platform design explanatory document [1, 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 4.1.3 gives an overview of the potential types of memory corruption vulnerabilities. The typical pitfalls during coding are detailed in section 4.1.4. In section 4.1.5 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 4.1.6.

4.1.3 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 [2]. The model (cf. Figure 4.1) summarizes the general causes of vulnerabilities, the way 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.

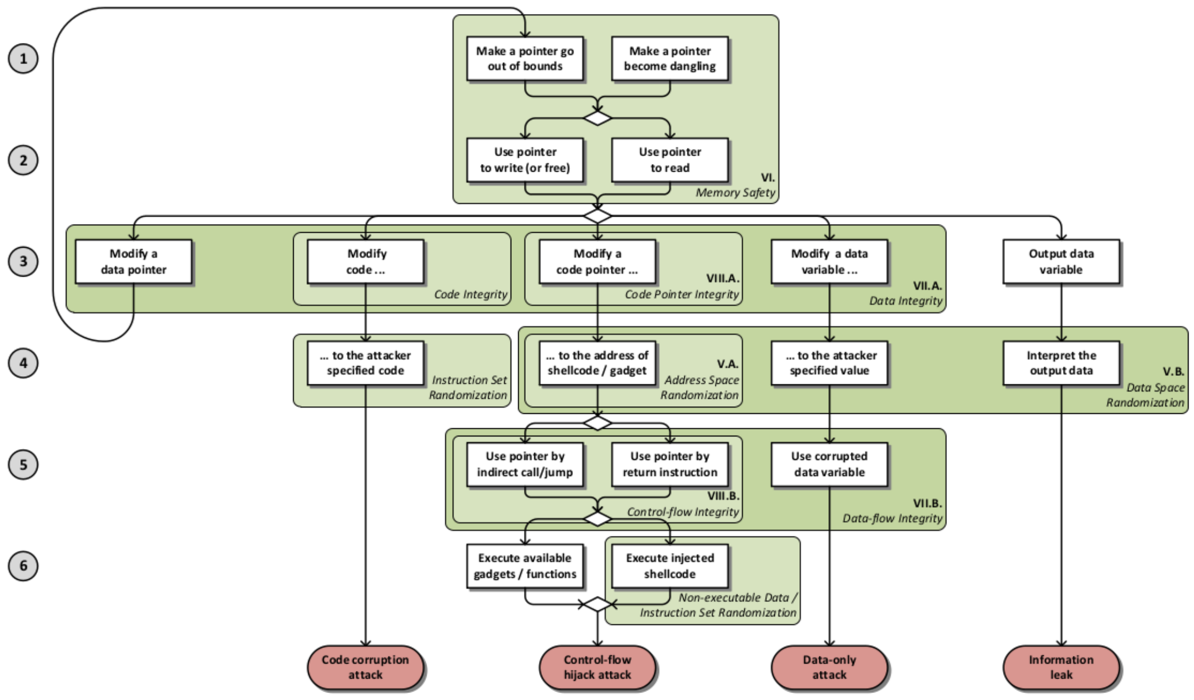
1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Figure 4.1: Attack model from [2] demonstrating four attack types, policies mitigating the attacks in different attack stages

4.1.4 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

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

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.

To avoid memory leaks, dangling pointers, or multiple deallocations of memory, usage of the “resource acquisition is initialization” (RAII) idiom is strongly encouraged. The goal is to make sure every previously allocated resource will be deallocated properly. Applying RAII to C++ means to encapsulate the allocation and corresponding deallocating of a resource into a dedicated class. While the resource will be allocated in the constructor, it’s meant to be deallocated by the destructor of the same object. This ensures proper resource deallocation at the end of the objects lifetime. However, it must be made sure the encapsulating objects lifetime is limited to the time the resource in question is in use. Usually, this is achieved by creating block scope objects placed on the stack. While RAII could also be used for memory allocations on heap, it is not limited to this use case. RAII also helps to prevent deadlocks when dealing with mutexes were the whole logic being protected by the mutex will be executed in context of the object acquiring and releasing a lock. It’s also noteworthy that RAII does not necessarily require an encapsulating class, but could also be implemented by means of compiler extensions were a stack object, e.g., a local variable or structure, could be associated with a corresponding clean up function. Such extensions are available for LLVM and other compilers like GCC supporting GNU syntax. This makes it possible to apply RAII to plain C.

In practice programmers should have coding guidelines at hand like the *MISRA* for safety-related systems. Unfortunately the C++ Core Guidelines do not cover a rule set for security related issues [3]. 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]. In any case the application of tools does not guarantee vulnerability-free code as runtime conditions (or execution contexts) are out-of-scope of such tools. Similarly, identified vulnerabilities may not be fixed sufficiently with respect to runtime behavior.

4.1.5 Attacks and Countermeasures

For a second line of defense it is assumed that vulnerabilities are present and that some will always be inserted by developers. Therefore a requirement for enhanced

[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`

countermeasures is independence from written code. The goal is to mitigate exploitation of vulnerabilities.

4.1.5.1 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 4.1).

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 [4, 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 [5] 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.

4.1.5.2 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 Integrity* policy is harmed (cf. Figure 4.1). 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 [6] 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 Programming* (ROP) [7], *Jump Oriented Programming* (JOP) [8], and *return-to-libc* [9], 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* (CPI) is satisfied if all dereferences that either dereference or access sensitive pointers, such as direct and references to code pointers, are not modified (cf. [10]). There are a few recent feasible approaches which detect the alteration of code pointers and references to code pointers at this early stage. The *Code Pointer Integrity* [10] 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) [10] is a relaxation of CPI.

Among others, CPS limits the set of protected pointers to code pointers (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* [11]. 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 targeting the return address, 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 execute 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 4.1.5.1. Moreover, the employed operating system must support it.

Code-reuse attacks are not affected by the W $\oplus$ X mechanism since existing memory regions marked as executable are utilized and no additional code must be injected into 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) [12]. Therefore, 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 Leaks (Section 4.1.5.4) 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) [13]. 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 overwritten by W $\oplus$ X.

4.1.5.3 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.

4.1.5.4 Information Leak

Memory corruption attacks are also used to leak memory contents. Therewith probabilistic countermeasures 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.

4.1.6 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 stated and which technical and security requirements are expected.

4.1.6.1 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 [14, p. 56]	No-eXecution bit (NX-bit), Page Table
	Intel64	eXecute Disable (XD-bit), Page Table
ARM	ARMv6	execute never bit (XN-bit), Page Table
	ARMv8-A	PEN privileged execute never, PAN privileged access never
SPARC	Oracle SPARC Architecture 2011	Translation Storage Buffer (optional)
PowerPC	IBM PowerISA [15] [16, p. 33]	Segment Lookaside Buffer (SLB)

Table 4.1: Examples of Hardware Support for Execution Prevention

Family	Name	Implementation
Linux	Linux kernel	Linux kernel mainline $\geq$ 2.6.8 [17]
	PaX	Patch for the Linux kernel, uses hardware support or emulates memory page permission [18]
	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
	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

Table 4.2: Examples of Operating System Support for Execution Prevention

4.1.6.2 Stack Smashing Protection (SSP)

System Level: Compiler

[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
[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

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.
	`-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.

[14]https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Instrumentation-Options.html#Instrumentation-Options
[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

	-protect_stack_all	The -protect_stack_all option adds this protection to all functions regardless of their vulnerability.

Table 4.3: Examples of Stack Smashing Protection Compiler Support

4.1.6.3 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 4.1.5.4) 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 allows to use 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
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]

Table 4.4: Examples of ASLR Operating System Support

[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

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.
	`-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.

[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`
[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`

Table 4.5: Examples of ASLR Compiler Support

4.1.6.4 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
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.

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

Table 4.6: Examples of CFI Compiler Support

4.1.6.5 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 [19].

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.

Table 4.7: Examples of CPI and CPS Compiler Support

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

4.1.6.6 Pointer Authentication

System Level: Hardware, Compiler

Pointer Authentication uses cryptographic functions to authenticate and verify pointers before dereferencing [11]. 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`.

Table 4.8: Examples of Pointer Authentication Compiler Support

4.1.7 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 4.1.2 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.

4.1.8 Horizontal Isolation

4.1.8.1 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 4.1.10 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 [20] 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 neighbouring 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. [21]) 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 [22].

4.1.9 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).

Table 4.9: Operating System Virtualization / Container Implementations

4.1.10 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 capabilities of a process, i.e. restricting what a process can do[44]. The classic way to

[42] https://linuxcontainers.org/
[43] https://www.freebsd.org/doc/handbook/jails.html
[44] Not to be confused with "what a process does", as the behaviour of a process is changed by an attack.

do this is by dropping privileges as soon as they are not needed anymore, i.e. *privilege 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) [23]	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) [23]	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 [24, 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).

OpenBSD	Pledge [25]	This functionality can be used to limit the system calls a process can execute. 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.

Table 4.10: Sandboxing Mechanisms in Operating Systems