Secure Computing Environments
Memory, Compiler and Virtual Machines

Christian Barthel,
Christian Kawalar,
Daniel Schlager

30.01.2015
1 Introduction

2 Memory Protection
   - Problem
   - OpenBSD’s ASLR and W^X

3 Compiler Options
   - Stack Smashing Protection

4 Virtual Machines
   - Overview of Virtual Machines
     - Security in Virtual machines
   - Process Virtual Machines
     - Dalvik
   - A Closer Look at Android and their Security
     - Platform Security Architecture

5 Conclusion
A word of caution...
A word of caution...

Disclaimer

It is not our intent to show you how to break into computer systems!
A word of caution...
A word of caution...

But!

“While you do not know life, how can you know about death?”

—Confucius
1 Introduction

2 Memory Protection
   - Problem
   - OpenBSD’s ASLR and W^X

3 Compiler Options
   - Stack Smashing Protection

4 Virtual Machines
   - Overview of Virtual Machines
     - Security in Virtual machines
   - Process Virtual Machines
     - Dalvik
   - A Closer Look at Android and their Security
     - Platform Security Architecture

5 Conclusion
Memory Layout revisited

Each program has a virtual address space. The `<TEXT>` section contains assembly commands (e.g., `<ADD>`, `<SUB>`). The `<DATA>` section contains global variables. The `<HEAP>` is the place where dynamic allocated data is stored. (`malloc`) The `<HEAP>` is also used to map libraries, files or devices into the address space. The stack is the place for function calls. Almost everything is predictable. Most of the memory is write- and executable.
Memory Layout revisited

- Each program has a virtual address space.

Stack

Heap

Data

Text

MAX

0

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
Memory Layout revisited

- Each program has a virtual address space.
- The **TEXT** section contains assembly commands (e.g., ADD, SUB, ...)

```
MAX
Stack

Heap

Data

Text

0
```
Each program has a virtual address space.

- The TEXT section contains assembly commands (eg ADD, SUB, ...)
- The DATA section contains global variables

Stack

Heap

Data

Text

MAX

0
Memory Layout revisited

- Each program has a virtual address space.
- The **TEXT** section contains assembly commands (e.g., `ADD`, `SUB`, ...)
- The **DATA** section contains global variables
- The **HEAP** is the place where dynamic allocated data is stored. (`malloc`)
Each program has a virtual address space.

The **TEXT** section contains assembly commands (e.g., `ADD`, `SUB`, ...)

The **DATA** section contains global variables.

The **HEAP** is the place where dynamic allocated data is stored. (`malloc`)

The **HEAP** is also used to map libraries, files, or devices into the address space.
Each program has a virtual address space.

The **TEXT** section contains assembly commands (e.g., `ADD`, `SUB`, ...)

The **DATA** section contains global variables

The **HEAP** is the place where dynamic allocated data is stored. (`malloc`)

The **HEAP** is also used to map libraries, files or devices into the address space

The stack is the place for function calls
Each program has a virtual address space.

The **TEXT** section contains assembly commands (e.g., `ADD`, `SUB`, ...)

The **DATA** section contains global variables

The **HEAP** is the place where dynamic allocated data is stored. (`malloc`)

The **HEAP** is also used to map libraries, files, or devices into the address space

The stack is the place for function calls

Almost everything is predictable
Each program has a virtual address space.

The **TEXT** section contains assembly commands (e.g., `ADD`, `SUB`, ...)

The **DATA** section contains global variables.

The **HEAP** is the place where dynamic allocated data is stored. (`malloc`)

The **HEAP** is also used to map libraries, files or devices into the address space.

The stack is the place for function calls.

Almost everything is predictable.

Most of the memory is write- and executable.
Where’s the problem?

Always the same old story...
Where’s the problem?

Always the same old story...

- an attacker finds a bug which damages memory (Bufferoverflow, ...)

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
Where’s the problem?

Always the same old story...

- an attacker finds a bug which damages memory (Bufferoverflow, ...)

- he analyzes the side effects, maybe to inject code or gain other advantages
Where’s the problem?

Always the same old story...
- an attacker finds a bug which damages memory (Bufferoverflow, ...)
- he analyzes the side effects, maybe to inject code or gain other advantages
- Various attacks possible: Heap, Stack, Libraries, ..

```c
int dangerous(int *a) {
    char inputBuffer[100];
    ...
    gets(inputBuffer);
    ...
}
```
Where’s the problem?

Always the same old story...

- an attacker finds a bug which damages memory (Buffer overflow, ...)
- he analyzes the side effects, maybe to inject code or gain other advantages
- Various attacks possible: Heap, Stack, Libraries, ...

```c
int dangerous(int *a) {
  char inputBuffer[100];
  ...
  gets(inputBuffer);
  ...
}
```
Where’s the problem?

Always the same old story...
- an attacker finds a bug which damages memory (Buffer overflow, ...)
- he analyzes the side effects, maybe to inject code or gain other advantages
- Various attacks possible: Heap, Stack, Libraries, ..

```c
int dangerous(int *a) {
    char inputBuffer[100];
    ...
    gets(inputBuffer);
    ...
}
```
Where’s the problem?

Always the same old story...
- an attacker finds a bug which damages memory (Buffer overflow, ...)
- he analyzes the side effects, maybe to inject code or gain other advantages
- Various attacks possible: Heap, Stack, Libraries, ...

```c
int dangerous(int *a) {
    char inputBuffer[100];
    ...
    gets(inputBuffer);
    ...
}
```

Stack / Process Address Space:
- high addresses
- call to dangerous
- Return Address
- Local Variables...
- inputBuffer[100]
- low addresses
Where’s the problem?

Always the same old story...

- an attacker finds a bug which damages memory (Buffer overflow, ...)

- he analyzes the side effects, maybe to inject code or gain other advantages

- Various attacks possible: Heap, Stack, Libraries, ..

```c
int dangerous(int *a) {
    char inputBuffer[100];
    ...
    gets(inputBuffer);
    ...
}
```
Memory Protection

Some memory protection mechanisms within the realm of the C/C++ runtime environment are
Memory Protection

Some memory protection mechanisms within the realm of the C/C++ runtime environment are

- W^X
Memory Protection

Some memory protection mechanisms within the realm of the C/C++ runtime environment are

- W^X
- Address space layout randomization (ASLR)
W^X with static binaries
W^X with static binaries

- Statically compiled binaries have a simple memory layout
W^X with static binaries

- Statically compiled binaries have a simple memory layout
W^X with static binaries

- Statically compiled binaries have a simple memory layout
W^X with static binaries

- Statically compiled binaries have a simple memory layout.
- The stack has a signal trampoline, called `sigtramp`, which has to be executable.
**W^X with static binaries**

- Statically compiled binaries have a simple memory layout
- The stack has a signal trampoline, called *sigtramp*, which has to be executable
W^X with static binaries

- Statically compiled binaries have a simple memory layout
- The stack has a signal trampoline, called sigtramp, which has to be executable
- Separate sigtramp from the stack, give it its own page

Christian Barthel, Christian Kawalar, Daniel Schlager
W^X with dynamic libraries

stack, RWX
sigtramp, RWX
bss, RW-
data, RW-
text, R-X
null page
null page

shared libraries are mapped into the address space of a process. They include additional GOT and PLT tables which must be written during execution. GOT is the shared lib global offset table, and PLT is the shared lib procedure linkage table. Since the PLT needs to be written and executed, an additional conversion is necessary.
W^X with dynamic libraries

- shared libraries are mapped into the address space of a process
shared libraries are mapped into the address space of a process
W^X with dynamic libraries

- Shared libraries are mapped into the address space of a process

- Stack, RWX
- Sigtramp, RWX
- Libc, RWX
- Ld.so, RWX
- Bss, RW-
- Data, RW-
- Text, R-X
- Null page

- Stack, RW-
- Sigtramp, R-X
- Got, data, RW-
- Text, plt, R-X
- Got, data, RW-
- Text, plt, R-X

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
W^X with dynamic libraries

- **shared libraries are mapped into the address space of a process**
- They include additional **GOT** and **PLT** tables which must be written during execution
- **GOT** is the shared lib global offset table
- **PLT** is the shared lib procedure linkage table
W^X with dynamic libraries

- shared libraries are mapped into the address space of a process
- They include additional **GOT** and **PLT** tables which must be written during execution
- **GOT** is the shared lib global offset table
- **PLT** is the shared lib procedure linkage table
- Since the **PLT** needs to be written and executed, an additional conversion is necessary.
ASLR - Randomized Memory

- stack, R-X
- sigtramp, R-X
- got, data, RW-
- text, plt, R-X
- got, data, RW-
- text, plt, R-X
- bss, RW-
- data, RW-
- text, R-X
- null page
ASLR - Randomized Memory

- Stack, R-X
- Sigtramp, R-X
- Got, data, RW-
- Text, plt, R-X
- Got, data, RW-
- Text, plt, R-X
- Text, plt, R-X
- Bss, RW-
- Data, RW-
- Data, RW-
- Text, R-X
- Text, R-X
- Text, R-X
- Null page
- Null page

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
ASLR - Randomized Memory
1. Introduction

2. Memory Protection
   - Problem
   - OpenBSD’s ASLR and W^X

3. Compiler Options
   - Stack Smashing Protection

4. Virtual Machines
   - Overview of Virtual Machines
     - Security in Virtual machines
   - Process Virtual Machines
     - Dalvik
   - A Closer Look at Android and their Security
     - Platform Security Architecture

5. Conclusion

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
What is a Stack Smashing Attack?

Listing 1: Vulnerable Function

```c
void vulnerableFunction(char *string) {
    char buffer[200];
    //BAD!
    //No size check!
    strcpy(buffer, string);
}
```
What is a Stack Smashing Attack?

- The stack contains the buffer, the return address and the parameters of the function.
What is a Stack Smashing Attack?

- With injected code, data gets overwritten.
Stack Overflow Attack

- To make use of a bufferoverflow, code (ie payload) can be injected.
Stack Overflow Attack

- To make use of a bufferoverflow, code (ie payload) can be injected.
- The Payload consist of three parts:
Stack Overflow Attack

- To make use of a bufferoverflow, code (ie payload) can be injected.
- The Payload consist of three parts:
  - Most CPUs have a NOP instruction (**no operation**): the instruction does nothing but increasing the Instruction Pointer by one.
Stack Overflow Attack

- To make use of a buffer overflow, code (ie payload) can be injected.
- The Payload consist of three parts:
  - Most CPUs have a NOP instruction (**no operation**): the instruction does nothing but increasing the Instruction Pointer by one.
  - We insert shellcode that, most of the time, opens a (root) shell.
Stack Overflow Attack

- To make use of a buffer overflow, code (ie payload) can be injected.
- The Payload consist of three parts:
  - Most CPUs have a NOP instruction (*no operation*): the instruction does nothing but increasing the Instruction Pointer by one.
  - We insert shellcode that, most of the time, opens a (root) shell.
  - Finally we set the RA (Return Address) back to a NOP instruction (guess the jump distance).
The stack contains the NOP instructions, our payload and the altered return address.
Stack Overflow Attack

- The stack contains the NOP instructions, our payload and the altered return address.
- We insert a bunch of NOP instructions to increase the chance of finding the right position.
Prevent the attack with GCC

- As we’ve already seen, the success of such attacks is more unlikely with ASLR.
Prevent the attack with GCC

- As we’ve already seen, the success of such attacks is more unlikely with ASLR.
- Now I will show you how to prevent such attacks with GCC and the `-fstack-protector` flag.
Prevent the attack with GCC

- Normally our subroutine would look like this:
Prevent the attack with GCC

- Normally our subroutine would look like this:
  - Initialization: The preparation of space on the stack for local variables.
  - Subroutine body: The subroutine’s implemented algorithm.
  - Clean-up: Removing local variables from the stack.
  - Return: Jump back to the original address before the branch.
Prevent the attack with GCC

- Subroutine code with SSP:
  - SSP’s prolog
  - Initialization: The preparation of space on the stack for local variables.
  - Subroutine body: The subroutine’s implemented algorithm.
  - Clean-up: Removing local variables from the stack.
  - SSP’s epilog
  - Return: Jump back to the original address before the branch.
Prevent the attack with GCC
Prevent the attack with GCC

- The canary got saved before the Return Address.
Prevent the attack with GCC

- The canary is a randomly generated number.
Prevent the attack with GCC

- The canary is a randomly generated number.
- GCC adds code at compile time, the code generates a random canary which will be checked after strcpy.
Prevent the attack with GCC

- The canary is a randomly generated number.
- GCC adds code at compile time, the code generates a random canary which will be checked after strcpy.
- It’s almost impossible to guess the actual canary, so there is no way to overwrite the canary in the payload with the right value.
Prevent the attack with GCC

Listing 4: -fno-stack-protector

vulnerableFunction:
.LFB2:
; Reserve Space on the stack

leaq -4352(%rsp), %rsp
orq $0, (%rsp)
leaq 4128(%rsp), %rsp

; Arguments from Register onto Stack
movq %rdi, −216(%rbp) ; 1st arg from rdi to stack
; Params for strcpy
movq −216(%rbp), %rdx ; 1st arg to rdx
leaq −208(%rbp), %rax ; 2nd arg to rax
; Call strcpy
movq %rdx, %rsi ; src address from rdx to rsi
movq %rax, %rdi ; dest address from rax to rdi
call strcpy@PLT ; call strcpy() @PLT
Listing 5 : -fstack-protector

```assembly
vulnerableFunction:
leaq  −4352(%rsp) , %rsp  ; Reserve space on stack
orq  $0 , (%rsp)
leaq  4128(%rsp) , %rsp

movq  %rdi , −216(%rbp)  ; 1st arg from rdi to stack
    ; SSP prolog, put canary to stack
movq  %fs:40 , %rax       ; canary from %fs:40 to ras
movq  %rax , −8(%rbp)    ; canary from rax to stack
xorl %eax , %eax         ; set rax to zero
    ; Params for strcpy
movq  −216(%rbp) , %rdx  ; 1st argument to rdx
leaq  −208(%rbp) , %rax  ; 2nd argument to rax
    ; Call strcpy
movq  %rdx , %rsi         ; src address from rdx to rsi
movq  %rax , %rdi         ; dest address from rax to rdi
    ; call strcpy()
    ; SSP epilog
movq  −8(%rbp) , %rax    ; canary from stack to rax
xorq %fs:40 , %rax       ; original canary XOR rax
je  .L2                  ; no overflow > xor == 0, jump
    ; Call stack_chk_fail@PLT ; overflow > xor != 0, kill
```
1. Introduction

2. Memory Protection
   - Problem
   - OpenBSD’s ASLR and W^X

3. Compiler Options
   - Stack Smashing Protection

4. Virtual Machines
   - Overview of Virtual Machines
     - Security in Virtual machines
   - Process Virtual Machines
     - Dalvik
   - A Closer Look at Android and their Security
     - Platform Security Architecture

5. Conclusion

Christian Barthel, Christian Kawalar, Daniel Schlager
Secure Computing Environments
Overview of Virtual Machines

A virtual Machine (VM) is an emulation of a particular computer system. Classification of virtual machines can be based on the degree to which they implement functionality of targeted real machines.

- System Virtual Machines (also known as full virtualization VMs)
- Process Virtual Machines
  - An example of Process Virtual Machines is Java virtual machine (JVM), Microsoft Common Language Runtime (CLR) and Dalvik
Overview of Virtual Machines

- A virtual Machine (VM) is an emulation of a particular computer system.
Overview of Virtual Machines

- A virtual Machine (VM) is an emulation of a particular computer system.
- Classification of virtual machines can be based on the degree to which they implement functionality of targeted real machines.
Overview of Virtual Machines

- A virtual Machine (VM) is an emulation of a particular computer system.
- Classification of virtual machines can be based on the degree to which they implement functionality of targeted real machines.
- System Virtual Machines (also known as full virtualization VMs)
Overview of Virtual Machines

- A virtual Machine (VM) is an emulation of a particular computer system.
- Classification of virtual machines can be based on the degree to which they implement functionality of targeted real machines.
- System Virtual Machines (also known as full virtualization VMs)
- Process Virtual Machines
Overview of Virtual Machines

- A virtual Machine (VM) is an emulation of a particular computer system.
- Classification of virtual machines can be based on the degree to which they implement functionality of targeted real machines.
- System Virtual Machines (also known as full virtualization VMs)
- Process Virtual Machines
- An example of Process Virtual Machines is Java virtual machine (JVM), Microsoft Common Language Runtime (CLR) and Dalvik
Security in Virtual machines

A virtual machine provides the following security features by default:

- memory management
- type safety
- exception handling
- garbage collection
- security and thread management

Christian Barthel, Christian Kawalar, Daniel Schlager
Security in Virtual machines

- A virtual machine provides the following security features by default:
Security in Virtual machines

- A virtual machine provides the following security features by default:
  - memory management
Security in Virtual machines

- A virtual machine provides the following security features by default:
  - memory management
  - type safety
Security in Virtual machines

- A virtual machine provides the following security features by default:
  - memory management
  - type safety
  - exception handling
Security in Virtual machines

- A virtual machine provides the following security features by default:
  - memory management
  - type safety
  - exception handling
  - garbage collection
A virtual machine provides the following security features by default:

- memory management
- type safety
- exception handling
- garbage collection
- security and thread management
Dalvik is the process virtual machine in Android. Programs are commonly written in Java and compiled to bytecode. Dalvik uses just-in-time (JIT) compilation. Android Runtime (ART) replaces the Dalvik Virtual Machine. ART uses the same input bytecode as Dalvik. The use of ahead-of-time (AOT) compilation (at installation) quickened dex.

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
Dalvik

- Dalvik is the process virtual machine in Android.
Dalvik

- Dalvik is the process virtual machine in Android.
- Programs are commonly written in Java and compiled to bytecode.
Dalvik

- Dalvik is the process virtual machine in Android.
- Programs are commonly written in Java and compiled to bytecode.
- Dalvik uses just-in-time (JIT) compilation
Dalvik

- Dalvik is the process virtual machine in Android.
- Programs are commonly written in Java and compiled to bytecode.
- Dalvik uses just-in-time (JIT) compilation
- Android Runtime (ART) replaces the Dalvik Virtual Machine.
Dalvik

- Dalvik is the process virtual machine in Android.
- Programs are commonly written in Java and compiled to bytecode.
- Dalvik uses just-in-time (JIT) compilation.
- Android Runtime (ART) replaces the Dalvik Virtual Machine.
- Use of ahead-of-time (AOT) compilation (at installation)
Dalvik

- Dalvik is the process virtual machine in Android.
- Programs are commonly written in Java and compiled to bytecode.
- Dalvik uses just-in-time (JIT) compilation
- Android Runtime (ART) replaces the Dalvik Virtual Machine.
- use of ahead-of-time (AOT) compilation (at installation)
- ART uses the same input bytecode as Dalvik.
System and kernel security

The foundation of the Android platform is the Linux kernel. The Linux kernel provides Android with several key security features, including:

- User-based permission model
- Process isolation
- Extensible mechanism for secure IPC
- The ability to remove unnecessary and potentially insecure parts of the kernel

Christian Barthel, Christian Kawalar, Daniel Schlager

Secure Computing Environments
System and kernel security

■ The foundation of the Android platform is the Linux kernel.
System and kernel security

- The foundation of the Android platform is the Linux kernel.
- The Linux kernel provides Android with several key security features, including:
The foundation of the Android platform is the Linux kernel.

The Linux kernel provides Android with several key security features, including:

- User-based permission model
System and kernel security

- The foundation of the Android platform is the Linux kernel.
- The Linux kernel provides Android with several key security features, including:
  - User-based permission model
  - Process isolation
**System and kernel security**

- The foundation of the Android platform is the Linux kernel.
- The Linux kernel provides Android with several key security features, including:
  - User-based permission model
  - Process isolation
  - Extensible mechanism for secure IPC
System and kernel security

- The foundation of the Android platform is the Linux kernel.
- The Linux kernel provides Android with several key security features, including:
  - User-based permission model
  - Process isolation
  - Extensible mechanism for secure IPC
  - The ability to remove unnecessary and potentially insecure parts of the kernel
Application security
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
- Application-defined and user-granted permissions
- Android Permission Model: Accessing Protected APIs
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
- Application-defined and user-granted permissions
- Android Permission Model: Accessing Protected APIs
  - Camera functions
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
- Application-defined and user-granted permissions
- Android Permission Model: Accessing Protected APIs
  - Camera functions
  - Location data (GPS)
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
- Application-defined and user-granted permissions
- Android Permission Model: Accessing Protected APIs
  - Camera functions
  - Location data (GPS)
  - Bluetooth functions, Telephony functions, SMS/MMS
All applications run in an Application Sandbox. They can only access a limited range of system resources.

Application-defined and user-granted permissions

Android Permission Model: Accessing Protected APIs

- Camera functions
- Location data (GPS)
- Bluetooth functions, Telephony functions, SMS/MMS
- Network/data connections
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.

- Application-defined and user-granted permissions

- Android Permission Model: Accessing Protected APIs
  - Camera functions
  - Location data (GPS)
  - Bluetooth functions, Telephony functions, SMS/MMS
  - Network/data connections

- Applications have to use APIs, they cannot access directly
Application security

- All applications run in an Application Sandbox. They can only access a limited range of system resources.
- Application-defined and user-granted permissions
- Android Permission Model: Accessing Protected APIs  
  - Camera functions  
  - Location data (GPS)  
  - Bluetooth functions, Telephony functions, SMS/MMS  
  - Network/data connections
- Applications have to use APIs, they cannot access directly
- Application signing
How to implement Security

Applications statically declare the permissions they require. Android system prompts the user for consent at the time the application is installed. There is no mechanism for granting permissions dynamically (at run-time).
How to implement Security

- Applications statically declare the permissions they require
How to implement Security

- Applications statically declare the permissions they require
- Android system prompts the user for consent at the time the application is installed
How to implement Security

- Applications statically declare the permissions they require
- Android system prompts the user for consent at the time the application is installed
- No mechanism for granting permissions dynamically (at run-time)
How to implement Security

- Applications statically declare the permissions they require
- Android system prompts the user for consent at the time the application is installed
- no mechanism for granting permissions dynamically (at run-time)
- in AndroidManifest.xml, add one or more uses-permissions tags
How to implement Security

Listing 6: AndroidManifest.xml

```xml
<permissions>
  <permission name="android.permission.CAMERA" />
  <group gid="camera" />
</permission>
<permissions>
  <permission name="android.permission.BLUETOOTH" />
  <group gid="net bt" />
</permission>
</permissions>
```
Conclusion

- Protecting computer systems from crackers is difficult
Conclusion

- Protecting computer systems from crackers is difficult
- People have found ways to get around all those security features we presented, especially in isolation
Conclusion

- Protecting computer systems from crackers is difficult.
- People have found ways to get around all those security features we presented, especially in isolation.
- Nevertheless, it’s much harder to break into a computer system with all those fancy security features.
Conclusion

- Protecting computer systems from crackers is difficult
- People have found ways to get around all those security features we presented, especially in isolation
- Nevertheless, it’s much harder to break into a computer system with all those fancy security features
- It’s difficult to find a good path between all those conflicting goals: comfort, security, performance, clean and simple implementation...