A tool flow and architecture for composable software protection

prof. Bjorn De Sutter
Computer Systems Lab
Ghent University

Code and data protection day, Paris-Saclay, 13 Dec 2018
Overview

• ASPIRE project introduction
• reference architecture for software protection
• compiler tool chain for software protection
• attack modeling
Data Hiding
Algorithm Hiding
Anti-Tampering
Remote Attestation
Renewability

SafeNet use case
Gemalto use case
Nagravision use case

ASPIRE Framework
Decision Support System
Software Protection Tool Chain

Protected SafeNet use case
Protected Gemalto use case
Protected Nagravision use case

https://www.aspire-fp7.eu
Man At The End (MATE) Attacks on Mobile Apps
Man At The End Attacks on Mobile Apps

- software analysis & editing tools
- FPGA sampler
- oscilloscope
- developer boards
- screwdrivers
- JTAG debugger
Economics of MATE Attacks

€/day

engineering
a.k.a. identification

exploitation

time

protection
Economics of MATE Attacks

- Protection
- Exploitation
- Diversification
Economics of MATE Attacks

€/day

engineering
a.k.a. identification

exploitation

time

protection

diversity

renewability
Attack Scope

- reverse engineering & tampering
- static attacks
  - structural code and data recovery (e.g., disassembly, CFG reconstruction)
  - structural matching of binaries
    - against known code (e.g., library identification)
    - of related binaries (e.g., diffing)
    - tampering (e.g., code editing)
- dynamic attacks
  - attacks on communication channels (e.g., sniffing, spoofing, replay attacks)
  - fuzzing, tracing, profiling, instrumentation, emulation
  - debugging (software or hardware debugger)
  - structure and data analysis (e.g., unpacking, taint analysis)
  - tampering (e.g., code injection, custom emulation, custom OS)
- hybrid attacks (e.g., concolic execution, static analysis on dynamic graphs)
Attack Models

start of the attack

sub-goal

final goal

attack steps
Reference Architecture

mobile device (untrusted, MATE attack)

client-side app
- hidden data
- hidden algorithms
- anti-tampering mechanisms

wireless/mobile network (untrusted, MITM attack)

secure channel

server (trusted)

server-side logic
- remote verifier
- bytecode provider
- renewability protection engine

client-side app
- renewability-supporting virtual machine
- remote attester

target platform: ARMv7-A / Android 4.4
native binaries / dynamically linked libraries
Plugin-based Tool Flow

C code
- annotated source code

ASPIRE source level protection
- data hiding
- algorithm hiding
- anti-tampering

Partially protected source code

Standard compiler

Object code

ASPIRE binary level protection
- data hiding
- algorithm hiding
- anti-tampering

ASPIRE protected program
- client-side app
- server-side logic

Available at https://github.com/aspire-fp7/

Client-side app

Server-side logic

Available at https://github.com/diablo-rewriter/
Decision Support System

input provided by the user
- platform description
- annotations
- assets

ASPIRE Decision Support System
- ASPIRE Knowledge Base

tool chain instructions
Industrial Use Cases

App
(Dalvik Java)

Kc

Android Media/DRM Framework

DRMPlugin
(dynamically linked C/C++ library)

Verify()

CryptoPlugin
(dynamically linked C/C++ library)

Decrypt()
Reference Architecture

- Data obfuscations
- White box cryptography (static keys, dynamic keys, time-limited)

```
ciphertext = AES_enc(plaintext, key);  \Rightarrow \quad \text{obf_key = receive(server);}
ciphertext = AES_WBC_enc(plaintext);
```

Legend:
- Source-to-source rewriting
- Binary rewriting
- Combination
Aspire Reference Architecture

- Data Hiding
- Algorithm Hiding
- Anti-Tampering
- Remote Attestation
- Renewability

- control flow obfuscations
- multithreaded crypto
- instruction set virtualization
- code mobility
- self-debugging
- client-server code splitting

legend:
- source-to-source rewriting
- binary rewriting
- combination
Reference Architecture

- code guards
- static and dynamic remote attestation
- reaction mechanisms
- client-server code splitting

Legend:
- source-to-source rewriting
- binary rewriting
- combination
Aspire Reference Architecture

Data Hiding | Algorithm Hiding | Anti-Tampering | Remote Attestation | Renewability

- native code diversification
- bytecode diversification
- renewable white-box crypto
- mobile code diversification
- renewable remote attestation

legend: source-to-source rewriting
        binary rewriting
        combination
Reference Architecture
A detailed description of each step depicted in Figure 6 is presented below.

1. The original application transfers control to the stub.
   Details: Currently this is implemented as an unconditional jump into the first part of the stub. Conceptually but not yet implemented this jump could be removed by Diablo by means of branch forwarding, so that the stub is inlined in the application code.

2. The stub sets up state for VM and transfers control.
   Details: The stub collects the contents of the physical ARM processor registers and calls the VM, passing the address of the corresponding bytecode (VM-image) as argument.
   When different stubs have different entry points into the VM, those entry points can be inlined in the stubs as well.

3. The VM fetches the bytecode and interprets it.
   Details: In case the bytecode is stored in encrypted form, the VM will need to decrypt it during this process.

4. After interpretation is finished, control is transferred to second part of the stub.
   Details: The bytecode comprises code to calculate the address where the native execution should continue. This address and the updated register values are returned to the stub.

5. The stub cleans up and transfers control back to the application.
   Details: The stub updates the physical ARM registers with the values the VM returned and jumps to the continuation address, transferring control back to the application.
Reference Architecture – Client-Server Splitting

Figure 9 – Structure of a message

3.3.7 Client/server code splitting

splitting sequence diagram

Figure 10 comprises the sequence diagram of the protection technique, followed by a detailed description of each step depicted. The figure depicts a prototypical execution of the protected application, where client:Client represents the client, while backendDispatcher:Server represents the slice manager that handles connections and messages, and slicedCode:Server is the sliced code at the server side.

Figure 10 – Sequence Diagram for Code Splitting

Seq#
Operation description
1
The protected client starts and sends a bootstrap message to the server.

Details: The client (labelled client:Client in Figure 10) starts its execution and sends a
attestators:
- code guards
- timing
- IO of functions
- control flow tags

verification:
- local vs. remote
- prevent replay attacks

delay reaction:
- attacker sees symptom
- hide relation with cause!

reaction:
- abort
- corruption
- notify server (block player)
- graceful degradation
- lower quality
Anti-Debugging through Self-Debugging
Anti-Debugging through Self-Debugging
Anti-Debugging through Self-Debugging
Anti-Debugging through Self-Debugging

process 1045

function 1

function 2a

function 3

mini debugger

debuggee

process 3721

function 1

function 2b

function 3

mini debugger

debugger
Aspire

Plugin-based Tool Flow

C code
annotated source code

ASPIRE source level protection
- data hiding
- algorithm hiding
- anti-tampering

partially protected source code

standard compiler

object code

ASPIRE protected program
- client-side app
- server-side logic

ASPIRE binary level protection
- data hiding
- algorithm hiding
- anti-tampering
- remote attestation
- renewability
- security libraries
void g(int x)
{
    _Pragma("ASPIRE begin softvm(softvm)"
    _Pragma("ASPIRE begin protection(obfuscations,
            enable_obfuscation(opaque_predicates:percent_apply=25))")
    int z=(x+x)^2;
    z = z*x;
    z = f(z);

    _Pragma("ASPIRE end") // obfuscations
    _Pragma("ASPIRE end") // softvm

    return z;
}
static const char cipher[]
    __attribute__((ASPIRE("protection(wbc,label(ExFix),role(input),size(16))"))))
    = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };

static const char key[]
    __attribute__ ((ASPIRE("protection(wbc,label(ExFix),role(key),size(16))"))))
    = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
        0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff };

char plain[16] __attribute__ ((ASPIRE("protection(wbc,label(ExFix),role(output),size(16))"))));

Pragma ("ASPIRE begin protection(wbc,label(ExFix),algorithm(aes),mode(ECB),operation(decrypt))")
decrypt_aes_128(cipher, plain, key);
Pragma("ASPIRE end")
Plugin-based Tool Flow

SLP03.01 WBC annotation extraction

SLC03.02 Parameters XML

SLP03.02 White-box tool python

SC04.01 .cl.h

SLP03.03 WBC header incl.

SC03 .cl.h

SLP03.06 WBC renewability

SC04.02 .cl.h
Plugin-based Tool Flow

D05.01 analysis results (aliasing, slices, ...)

SLP05.01 data obfuscation
TXL

SC05 .i

SLP05.02 source code analysis
CodeSurfer

SC06 .i
Plugin-based Tool Flow

- **D01**: Annotation facts
- **D02**: Map file (a.out.map | liba.so.map)
- **BC02**: Binary | Library (a.out | liba.so)
- **BC08**: Object code (.o)

**Extractable chunks**

- **BLP01.01**: Bytecode chunk identifier (diablo)
- **BLP01.02**: Instruction selector (.so)

**Linker script**

- **BLC02**: Extractable chunks (JSON)
- **BLP02**: X-translator (...)

**BC03**: Bytecode + stubs (.o)
Attack Modeling

- experiments with professional hackers
- public challenge for amateurs
- methodological analysis of reports

M. Ceccato, P. Tonella, C. Basile, P. Falcarin, M. Torchiano, B. Coppens, B. De Sutter
*Understanding the Behaviour of Hackers while Performing Attack Tasks in a Professional Setting and in a Public Challenge*
Empirical Software Engineering, 2018
Attack Taxonomy

- Asset
- Attack strategy
- Background knowledge
  - Knowledge on execution environment framework
- Workaround
- Analysis / reverse engineering
  - Static analysis
    - Differing
    - Control flow graph reconstruction
  - Dynamic analysis
    - Dependency analysis
      - Data flow analysis
    - Memory dump
    - Monitor public interfaces
    - Debugging
- Obstacle
- Protection
  - Obfuscation
    - Control flow flattening
    - Opaque predicates
    - Virtualization
  - Anti-debugging
  - White box cryptography
    - Tamper detection
      - Code guard
        - Checksum
    - Execution environment
      - Limitations from operating system
- Weakness
  - Global function pointer table*
  - Recognizable library

* indicates multiple inheritance; new concepts added during the second qualitative experiment are underlined; concepts emerged in both experiments are in boldface.

Fig. 3: Taxonomy of extracted concepts (part I): the analysis methods and tools hackers may use (Analysis / reverse engineering), weaknesses in design and coding of the application to protect that may help the hacker tasks (Weakness), the difficulty hackers may experience when trying to perform an attack task (Difficulty), the protections a defender can place to limit certain attack steps (Obstacle), and other high-level concepts that characterize the hacking scenarios (Asset, Attack strategy, Background knowledge, Workaround).
Fig. 4: Taxonomy of extracted concepts (part II): the attack steps hackers may perform. They include the operations to prepare the attack (Prepare attack), the tasks to understand the software through reverse engineering the application code (Reverse engineer software and protections), the modifications to code and executions to tamper with the application (Tamper with code and execution), and the tasks to evaluate whether the attack was successful or not and learn from errors (Analyse attack result). * indicates multiple inheritance; new concepts added during the second qualitative experiment are underlined; concepts emerged in both experiments are in boldface.
3.3.2 How hackers build attack strategies

Figure 7 shows a model of how hackers come to the formulation and validation of hypotheses about protections, and how this eventually leads to the construction of their attack strategy. Hypothesis making requires (see "cause" relations in Figure 7) running (static / dynamic) program analyses and interpreting the results by applying background knowledge on how software protection and obfuscation typically work (e.g., [O:E:4] “static analysis to detect anti-debugging protections”). Identifying protections or libraries involved in protections is also an important prerequisite to be able to formulate hypotheses. When an attack attempt fails (see "condition for" relation on the left in Figure 7), the reasons for the failure often provide useful clues for hypothesis making (sentence “As the original process is already being ptraced, this prevents a debugger, which typically uses the ptrace system, from attaching”, annotate as [P:A:50] “Guess: avoid the attachment of another debugger”).

To confirm the previously formulated hypotheses, further analyses are run and interpreted based on background knowledge (see "cause" relations connected to Confirm hypothesis). Pattern matching is also useful to confirm hypotheses (e.g., [P:F:26] “Repeated execution patterns are identified and matched against repeated computations that are expected to be carried out by the relevant code”; [P:D:25] “mapping of observed (statistical) patterns to a priori knowledge about assumed functionality”). Another activity that contributes to the confirmation of previously formulated hypothesis is the creation of a small program that replicates the conjectured protection (e.g., [P:F:47] “Understanding is carried out on a simpler application having similar (anti-debugging) protection”).

Once hypotheses about the protections are formulated and validated, an attack strategy can be defined. This requires all the information gathered before, includ-
Attack Behavior Models

Once tools are selected and customized, they are used to find patterns, by running further analyses on the protected code, or they are used directly to undo protections and mount the attacks (see "used to" relations in the middle of Figure 8). When existing tools are insufficient for the hackers' purposes, new tools might be constructed from scratch. This is potentially an expensive activity, so it is carried out only if existing tools cannot be adapted for the purpose in any way and if alternative tools or attack strategies are not possible. One case where such tool construction from scratch tends to be cost-effective is when hackers want to execute a part of the software out of context, to better understand its protections (see "used to" relation connected to Out of context execution). In fact, this usually amounts to writing scaffolding code fragments that execute parts of the application or library under attack in an artificial, hacker-controlled, context (L:E:17 "write custom code to load-run native library").

The model in Figure 8 was fully applicable to the public challenge annotations, with no need for any extensions. The public challenge experiment provided substantial further support to the general validity of this model. The model shows that tools play a dominant role in the implementation of attacks. Hence, software protections should be designed and realized based on an amount of knowledge of tools and of their potential that should be as deep and sophisticated as the hackers' one. Preventing out of context execution is another important line of defence against existing and new tools.

3.3.4 How hackers defeat protections

The actual execution of an attack against a protection aims at defeating it, by bypassing it, building a workaround, undoing the protection completely, or overcoming it in some other way. Figure 9 shows a model of such activities.
Fig. 9: Model of hacker activities related to defeating protections by undoing, overcom-
ing, working around, or bypassing them. This means that instead of reversing the ef-
effect of a protection (e.g., deobfuscating the code), they gather enough infor-
mation to be able to manipulate the code and the execution so as to achieve the
desired effect, without having actually removed the protection. Gathering the infor-
mation and performing the manipulations with the protections still present typically
requires a considerable effort in analysis, and in building external tools, scripts, or tool
extensions. Overcoming a protection eventually relies on the possibility to alter the
normal flow of execution, this is the reason for a causal relation between
Tamper with execution and Overcome protection.

In some instances, altering the execution flow with external tools is not enough,
not possible, or requires too much effort. In such cases, hackers may write custom
workaround code (Build workaround) that is integrated with or replaces the existing
code, with the purpose of preserving the correct functioning of the software, while at
the same time making the protections ineffectual.

Sometimes hackers run program analyses to obtain information that is useful for
manually undoing protections. For instance, dynamic analysis and symbolic execution
can be used to understand if a predicate is (likely to be) an opaque one, such that one of
the two branches of the condition containing the predicate can be assumed to be dead
code that was inserted just to obfuscate the program ([L:F:2] "Undo protection (opaque
predicates) by means of dynamic analysis and symbolic execution"). The analyses
needed to undo protections may be quite sophisticated, hence requiring non trivial
tool customization (see incoming relations of Undo protection in Figure 9).

To overcome a previously identified protection, hackers alter the execution. For
instance, if they have identified some library calls used to implement a protection,
they may try to intercept such calls and replace their parameters on the fly; they may
skip the body of the called functions and return some forged values; or, they may
redirect the calls to other functions ([O:F:17] "Tamper with system calls (ptrace) that
implement the anti-debugging protection by means of an emulator"); see causal relation
to Overcome protection in Figure 9). To achieve the desired effect, this might require...
Questions?

The Aspire project has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.