Compilation using LLVM

Juan Manuel Martinez Caamaño (@jmmartinez)

Quarkslab

Last course

- Dissassembling
- Instrumentation

Today's objective

- Startup Functions
- Dynamic Protections

Dynamic Protections

Dynamic Analysis

Reverse engineers rely on dynamic attacks to overcome the limitations of static analysis

- Used to reduce the scope of static analysis

These include:

- Debugging
- Instrumentation
- Hooking functions
- Emulation
- Tampering the program

Dynamic Protections

Runtime verifications to ensure the program is running in a safe environment.

- Detect unexpected behaviours
- Detect debuggers
- Detect instrumentation
- Detect modifications in the code

Anti-*: Debug, Jailbreak, Root, ...

Anti-*

Debuggers alter the environment in which a program executes

- Special flags are setted
- Functions behave differently
- Dynamic libraries loaded in the process
- More instructions than expected are executed

Example GDB-ptrace

GDB relies on `ptrace` to control the executable

```
if(ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
  crash_application();
```

Example GDB-ptrace

GDB relies on `ptrace` to control the executable

```
if(ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
  crash_application();
if(ptrace(PTRACE_TRACEME, 0, NULL, NULL) == 0)
  crash_application();
```

Example Breakpoint detection

Software breakpoints introduced by debuggers are achieved by inserting a special instruction in a target function's code.

- `0xcc` in x86 and x86_64
- `0xd4200000` in arm64
- `0xbebe` in thumb
- `0xe7f001f0` in arm

Example Breakpoint detection

Difficult to implement:

- Variable instruction size
- Constant pools mixed in the code
- Splitting hot and cold function code
- Other obfuscations (e.g. random assembly)

Example Code Integrity

Compute a checksum of the code and verify against a reference value.

```
void foo() {
  ...
foo_end:
}

void check_foo() {
  uint32_t hash = crc32((int32_t*)foo, (int32_t*)foo_end);
  if(hash != 0xf00)
    exit(-1);
}
```

Example Code Integrity

Difficult to implement:

- Can't be implemented at the IR level, need help at the backend
- Circular checks
- Relocations

When to do the checks?

Startup Checks

Functions executed before the main function

- Low impact on performance
- The standard library might not be initialized
- Easy to identify

Periodically over the code

Mine the entire code with checks.
Manually dissabling the checks is too painful.

- Risk of injecting checks in hot code
- Risk of injecting checks that are never executed

Responding to a detection

Responding to a detection

Once an attack is detected the application must respond to it:

- Make the application slow
- Send a message to a server
- Stop the application

Stop the application

Corrupt the program state.
The error propagates and eventually the application crashes.

- Filp a bit from a variable
- Write into a buffer
- Jump into unexpected code
- Compute the value of a constant from the check result

Stop the application

Have enough distance from the check, the response, and the crash to not link them.

Stop the application

Erase the return addresses from the stack to forbid getting a stacktrace.

```
void crush() {
  int stack;
  for(int i = 0; i != 64; ++i)
    *(((int*)&stack)+i) = 0;
}
```

Builtins

Builtins

Some of the checks can be pretty complex. Implementing the code that generates the LLVM-IR is not a reasonable option.

Ideally we would implement them in a high level language.

So, how do we inject them in the code ?

External Library

Create a static library, and link every code with this library.

- Straightforward to implement
- Clear frontier between the builtin and the user's code

Pre-Compiled Builtins

Precompile the bultins and generate LLVM-IR. This LLVM-IR is loaded and linked with the current compilation unit.

- The user's code and the builtin can be optimized as a whole
- The target must be precisely known in advance (specially if using the C/C++ runtimes)

On-the-fly Compiled Builtins

The code is parsed and compiled as needed. LLVM-IR is generated and is linked with the current compilation unit.

- The user's code and the builtin can be optimized as a whole
- The LLVM is not designed for this usage

Conclusions

- Anti-* checks can disable tools used by reverse engineers
- Mine the code with checks
- Try to be as undetectable as possible, but also effectively stop the reverser from recovering useful information