

Technische Universität Darmstadt
Intel Collaborative Research Institute for Secure Computing (ICRI-SC)
Prof. Dr.-Ing. Ahmad-Reza Sadeghi
Dr.-Ing. Lucas Davi
M.Sc. Christopher Liebchen

Tutorial, 31 August 2016, 15:00 - 17:00

The Continuing Arms Race:

A Journey in the World of Runtime Exploits and Defenses

16th International School on Foundations of Security Analysis and Design (FOSAD)

Code-reuse attacks such as return-oriented programming constitute a powerful and prevalent attack class that is frequently leveraged to exploit software programs. The goal of this exercise is to construct such code-reuse attacks against vulnerable mobile Android applications.

First, you need to install VirtualBox¹ and import the 4GB tutorial image². On the Desktop of your Ubuntu 10.04 VirtualBox image, you will find a sample Android application project called *hello-jni*. This application leverages the Android NDK (Native Development Kit) to link to a shared library called *libhello-jni.so*. The Android application invokes the shared library to output the string *Hello from JNI!* on the user's display. Further, the shared library contains several native functions (written in the C language) to open and read a local file from the sd-card, called *input*, and store its content into a local buffer.

In the following Exercise No. 1 and No. 2, you need to identify the embedded vulnerability inside the Android application, and exploit the vulnerability to open the Android Web-Browser providing the URL <http://www.stackoverflow.com> as first argument. Due to the time limit, we included Exercise No. 3 as optional. In this exercise, you will construct several advanced exploits to drop a phone call.

To correctly construct the attacks, it is necessary to debug the application with the standard Linux debugger *gdb*. The debugger facilitates determining the correct memory addresses of functions and gadgets you desire to invoke in your code-reuse attack. In Appendix A, we added an excerpt of the *gdb* command reference.


Before you start constructing your code-reuse attacks, please make yourself comfortable in using and debugging the Android application.

¹<https://www.virtualbox.org/wiki/Downloads>


²https://drive.google.com/open?id=0BxRA3a6PD_VDaUx1T2hhemdoX2c


Android Emulator, Debugging, and File Transfer

To launch the sample application on the Android emulator, proceed as follows.

- a) Open a terminal
- b) Start Eclipse: `$ /home/android/eclipse/eclipse &`
(the hello-jni project is loaded automatically)
- c) Start the app: 
- d) Once the Android emulator is started, deactivate the screensaver to view the app's main activity.

To debug an Android native code application, proceed as follows.

- a) If not yet opened, open Eclipse as described above
- b) Start the app in debug mode: 
- c) Open a terminal
- d) Navigate to the project's folder: `$ cd Desktop/hello-jni/`
- e) Attach to the running app with gdb: `$../../android-ndk-r8/ndk-gdb`
- f) Ignore the error messages concerning the debug symbols of shared libraries

At this stage, you can use the gdb commands listed in Appendix A. For the sake of demonstrating the effects of breakpoints, set a breakpoint by executing `break vulnerable` in the gdb console. If you set a breakpoint in gdb, you need to (i) continue the execution in the gdb console via the `continue` command, and (ii) resume the execution in Eclipse by clicking on the Resume button: . Subsequently, you should see in your gdb console that the breakpoint has been reached. For now, quit the gdb console by executing the `quit` command and confirming that you quit a running program.

Since the Android application opens a file stored on the sd-card, we shortly describe the process of modifying the file and uploading it to the Android emulator. To change the content of the `input` file, proceed as follows.

- a) Open the `input` file in a hex editor: `$ ghex2 input`
- b) Enable the ghex2 insert mode via the ghex2 navigation bar.
- c) Replace the first four bytes with: `55 66 77 88`
- d) Always add an extra `FF` as last character of the file
- e) Save your changes

To transfer the `input` file between your host machine and your Android emulator's sd-card, execute the following shell commands. A prerequisite for executing these commands is that the Android emulator is running.

- Send file from host to emulator: `$ adb push input /sdcard/`
- (You can also send files from emulator to host: `$ adb pull /sdcard/input`)
- Open emulator shell to check for stored files: `$ adb shell`

Note that files are directly overwritten on the Android emulator. Hence, backup your files between the exercises to avoid losing your exploit payloads.

1 Return-into-Library Attack

In this exercise you need to construct a return-into-library exploit that invokes the subroutine *startBrowser()*. In order to construct the attack, proceed as follows:


- Identify and examine the buffer overflow vulnerability in the native code part
- Determine the number of bytes your `input` file requires to overwrite the return address
- Debug the application to determine the memory address of *startBrowser()*. Please note that although the application is compiled in THUMB mode, *gdb* displays disassembled THUMB code with even memory addresses. However, internally the application still executes THUMB code from odd memory addresses, i.e., always set the last significant bit when using THUMB code in your exploit.
- Construct your exploit by modifying the `input` file
- Upload the modified `input` file to your Android emulator and test your exploit

2 Basic Return-Oriented Programming Attack

In this exercise you need to construct a return-oriented programming attack that launches the browser with `http://www.stackoverflow.com` but **without** invoking the *startBrowser()* subroutine. In contrast, you have to generate the return-oriented programming payload based on the sequences provided in the *someArbitraryROPsequences()* subroutine. In order to construct the attack, proceed as follows:

- Identify the memory addresses of the sequences embedded in *someArbitraryROPsequences()*
- Identify the memory address of the string:
`am start -a android.intent.action.VIEW -d http://www.stackoverflow.com`
- Identify the memory address of the libc function *system()*
- Reverse-engineer the application to determine the ARM register where *system()* expects its first function argument
- Construct your exploit by modifying the `input` file and leveraging the sequences from *someArbitraryROPsequences()* and the libc function *system()*
- Upload the modified `input` file to your Android emulator and test your exploit

3 OPTIONAL: Advanced Return-Oriented Programming Attack

In this exercise, we construct several advanced exploits. In particular, we launch the browser with an attacker-chosen URL, and construct an exploit that drops a call. While you perform this exercise do not restart the emulator, and launch all the attacks by initiating a debug session, i.e., starting the application in debug mode . The reason is twofold: first, when you restart the emulator, the stack base address might change. Second, the stack base address differs from debug to run mode. This randomization did not affect your exploits in the preceding exercises, but impacts your exploits in this exercise as we leverage stack addresses.

In order to construct the exploits, proceed as follows:

- First, you should construct an attack that calls the browser with your own chosen URL. For this, you need to provide the entire Android command `am start -a android.intent.action.VIEW -d http://...` on the application's stack. This is achieved by (i) providing the target string in the `input` file, (ii) determining the stack address where the string is located in debug mode, and (iii) adjusting the exploit payload in the `input` file.
Hint: You do not need to determine the hexadecimal representation of your target string, because `ghex` allows you to enter ASCII characters.
- Next, instead of launching the browser, you should exploit the vulnerable application to dial a phone number of your choice. Compared to the last attack, you only need to change the string `android.intent.action.*` and provide the appropriate argument, i.e., your chosen telephone number. As an example, your new string could be as follows: `am start -a android.intent.action.DIAL -d tel:123456`
- As you will recognize, the above attack still requires the user to explicitly click on the dial button. However, Android offers the `android.intent.action.CALL` command to drop a call. Change your input file to use the `CALL` command and test your exploit. Find out why this attack fails by inspecting the Logcat messages in your Eclipse debug window.
- Change the `hello-jni` application accordingly to allow the phone call attack to succeed. Before doing any changes to the application either consult Luca or Christopher to check if you are performing the correct changes.

A GNU Debugger gdb

The following gdb commands may help you to solve this exercise:

- **quit**: Exit gdb
- **list**: Show lines of the source code
- **disassemble *function***: Disassemble a function
 - **disassemble main**: Disassemble the main function
- **break**: Set a breakpoint
 - **break *0x800000F**: Set a breakpoint at address **0x800000F**.
 - **break main**: Set a breakpoint at the **main** function
- **run [*arglist*]**: Run the program with arguments *arglist*
- **step instruction**: Execute one assembler instruction
- **continue**: Continue program execution until the next breakpoint is hit
- **info registers**: Print the processor register values.
- **display /3i \$pc**: Print on every step the next three instructions.
- **print [*expression*]**: Show value of *expression*
 - **print /x \$sp**: Print the value of the %sp register in in hexadecimal (**x**) notation
- **x/[*Nuf*] *expression***: Examine memory at address *expression*, whereas **N** indicates the Number of units to display, **u** the unit size, and **f** the printing format.
 - **x/4wx \$sp**: Print the top four (**4**) 32 bit words (**w**) on the stack (**\$sp**) in hexadecimal (**x**) notation
 - **x/1s \$r0**: Display one String (**1s**) starting from the address stored in (**\$r0**).