Chapter 7
Java Serialization

Dirk Pawlaszczyk

Abstract Java Serialization is a popular technique for storing object states in the Java programming language. In the field of mobile forensics, we come across such artefacts. App developers very often resort to this technique to make their application state persistent. Serialization is also used when transferring data over a network between two Java applications using Remote Method Invocation(RMI). In the past, there have been recurring security issues associated with this technology. Despite its importance for forensic casework, one can hardly find any literature on this topic. In this chapter, we give an insight into the binary format. For this purpose, special features of the format are presented using an example. In addition to the actual protocol structure, basic steps for acquiring such data and analyzing it will be discussed. Practical hints for searching serials are given. Finally, the security issues are addressed.

7.1 Introduction

Among app developers, the Java programming language has been the first choice for many years. The popularity of the language can be attributed to its simple syntax and compelling framework. As with any object-oriented language, the execution state of the program is managed through objects. From time to time, an application needs to back up its current state to disk. Of course, it is possible to store important data in a database such as SQLite. However, this usually requires object-relational mapping to be performed first. From the beginning, Java offers an alternative for persistent writing of objects: the so-called Java Object Serialization (JOS) [93]. Java's standard serialization seems to be a good choice, especially for app developers who want to store objects' current execution state. By serialization, we understand the ability to

Faculty for Applied Computer Science
University of Applied Sciences (Hochschule Mittweida), Technikumplatz 17, 09648 Mittweida, Germany, e-mail: pawlaszc@hs-mittweida.de

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_7

convert an object in the application's main memory into a format that allows the object to be written to a file or transported over a network connection.

Since many apps rely on this format by default to store their program data, investigators are of particular interest. This chapter will take a closer look behind the scenes at the Standard Serialization concept in Java. We pay special attention to the binary format and explore how to analyze this file type.

7.2 Object Serialization in Java

7.2.1 Serialization Techniques in Java

Under Java SE, objects can be automatically mapped and stored persistently using various approaches [91], [54]:

- **Standard serialization:** The object structure and states are saved in a binary format. As already mentioned, this procedure is also called Java Object Serialization (JOS). Standard serialization is very important for remote method calls and storing things over time and then retrieving them from the closet at some point.
- **XML serialization via JavaBeans Persistence:** JavaBeans - and only such - can be saved in an XML format. One solution is JavaBeans Persistence (JBP), which was originally intended for Swing. When the state of a graphical user interface is binary persisted with JOS, changes to the Swing API's internals are not easily possible since the binary format of JOS is very tightly coupled with the object model. That is, objects sometimes cannot be reconstructed from the binary document. JBP decouples this by communicating only through setters/getters and not on internal references, which are an implementation detail, which can change at any time. Nowadays, JBP hardly plays a role in practice.
- **XML mapping via JAXB:** With JAXB, a second API is available for mapping the object structure to XML documents. The eXtensible Markup Language (XML) supports a text-based data format based on markups. The platform-independent exchange format is part of the standard library from version 6. It is a fundamental technology, especially for Web service calls.

All three options are already built into Java by default. The standard object serialization creates a binary format and is very strongly oriented towards Java. Other systems cannot do much with the data. XML is convenient as a format because other systems can process it. Another compact binary format that also allows interoperability is Protocol Buffers [1] from Google. The company uses it internally when different applications are to exchange data.

Finally, objects can also be stored in relational databases called object-relational mapping (OR mapping). This technique is very sophisticated because the object

[1] http://code.google.com/p/protobuf/

models and tables are quite different. The Java SE does not offer any support for OR mapping, but it can be done with additional frameworks, such as the JPA (Java Persistence API).

7.2.2 Serialization by Example

The traditional way from an object to persistent storage is via Java's serialization mechanism [57][54]. JOS is the technology we want to deal with in the following. The standard serialization offers a simple possibility to make objects persistent and to reconstruct them later. The object state (no static ones!) is written into a byte stream (serialization). From this, it can be reconstructed to an object again later (deserialization). The object state is written into a serial data stream of 0 and 1. Java provides two special classes for this purpose: *ObjectOutputStream* and *ObjectInputStream* with a *writeObject()* respectively *readObject()*-method. Both classes can be found in the *java.io* package of the Java standard class library [2]. To save an object's state, we must pass the object reference as a parameter to the writeObject()-method. In the Java ecosystem, the applications programmers are encouraged to use serialization almost everywhere.

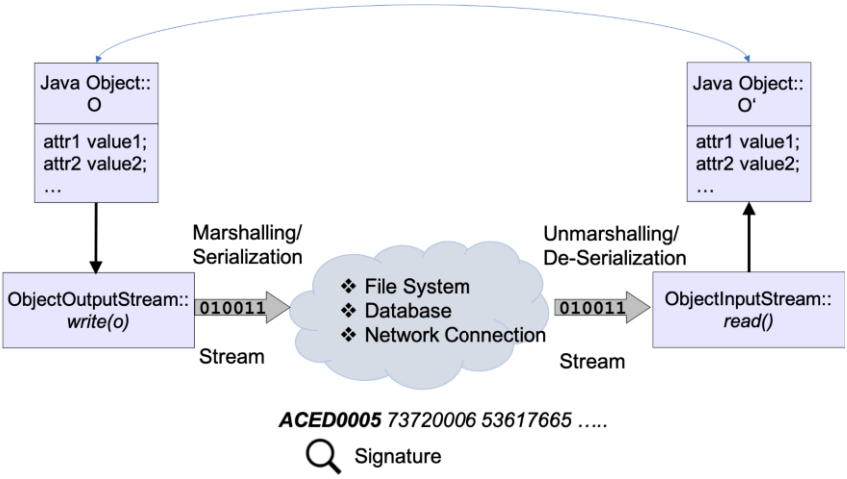


Fig. 7.1: Java Object Serialization (JOS) - Concept

Serialization concept can show its strengths, especially in communication between different Java processes distributed over a network. We serialize some of the objects, send them to another process for processing, serialize the transformed object and send

[2] https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html

it back. To illustrate this, we will discuss a small example program. In the following, we want to make a class *SaveMe* serializable. For this, we need the following code:

Listing 7.1: Class definition of a class to serialize

```
import java.io.Serializable;

public class SaveMe implements Serializable {

        private static final long serialVersionUID = 1L;
        private int x;
        private double d;
        private String s;

        public SaveMe() {
                this(100, 3.14, "hello");
        }

        public SaveMe(int x, double d, String s) {
                this.x = x;
                this.d = d;
                this.s = s;
        }

        public String toString() {
                return s + " " + x + " " + d;
    }
}
```

In Listing 1.1, the class *SaveMe* is defined first. In order for objects to be Serialized, the classes must implement the *Serializable* interface. This interface thus serves as a marker to indicate that the class can be Serialized.

! Attention

When serializing an object, only its attributes are stored. Methods or program code remains in the *.class* file.

Java is assigning a serial number to each object of the class it writes to a stream. This serial number is then used to re-create the class when it is reread. If two variables contain references to the same object and we write the objects to a file and later read them from the file, then the two objects that are read will again be references to the same object. All attributes of an object can be made persistent in this way. However, there are two exceptions. Attributes defined with the prefixed key *transient* are not Serialized. This identifier was explicitly introduced to exclude an attribute from Serialization. It can be helpful, for example, if confidential or volatile data should not be saved. The second exception is class attributes preceded by the keyword *static*. With one exception, such attributes shared by all objects of a class are not Serialized. In our example, there are no transient attributes and only one static class attribute. Since the serialVersionUID property is defined as

static and thus should not be stored. In this case, however, an exception is made. In Java, serialVersionUID is like version control, ensuring that both Serialized and deSerialized objects use the compatible class. For example, if an object is saved into a stream with serialVersionUID=1L, when we convert the stream back to an object, we must use the same serialVersionUID=1L. Otherwise, an InvalidClassException is thrown.

If we create a *SaveMe* object $o1$ and call $writeObject(o1)$, the ObjectOutputStream pushes the variable assignments (here x, d and s) into the data stream. An example is shown in the next listing:

Listing 7.2: Output Class 'Serializer'

```
import java.io.*;

public class Serializer {

public static void main(String[] args) throws IOExectpion {
                ObjectOutputStream o =
                    new ObjectOutputStream(
                            new FileOutputStream("saved.ser"));
                SaveMe o1 = new SaveMe();
                o.writeObject(o1);
                o.close();
}}
```

This routine creates a file, *saved.bin*, on the disk that contains the serialized object. With a few lines, the state of our object $o1$ can be saved to a file. In the example shown, the serialized object is written to a file. To send the object over the network, we have to create an object of the Socket class and start writing to the output stream of this class:

Listing 7.3: Output to a socket connection instead of a file

```
...
Socket connection = new Socket(hostName, portNumber);
OjbectOutputStream oos =  new
    ObjectOutputStream(connection.getOutputStream(), true);
oos.writeObject(o1);
```

We only need to adjust two lines in our program, and we are ready to go. It could not be simpler. However, this form of serialization also has disadvantages. Standard serialization works according to the principle: Everything reached from the object graph enters the data stream serialized. Suppose the object graph is extensive, the time for serialization and the data volume increase. Unlike other persistence concepts, it is not possible to write only the changes. For example, if only one attribute value has changed in an extended object list, the entire list must be rewritten. This is not efficient. However, let us focus on analyzing the binary format.

The output file <saved.ser> from the above example has a size of 80 bytes. The content of the file can be seen in Fig. 7.2. In addition to the actual attribute values, information about data type and class type is also stored in the file. Fig. 7.3 offers a high-level look at the serialization algorithm for this example. In the next section,

```
00  AC ED 00 05 73 72 00 06 53 61 76 65 4D 65 00 00 00    ....sr..SaveMe...
11  00 00 00 00 01 02 00 03 44 00 01 64 49 00 01 78 4C    ........D..dI..xL
22  00 01 73 74 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F    ..st..Ljava/lang/
33  53 74 72 69 6E 67 3B 78 70 40 09 1E B8 51 EB 85 1F    String;xp@...Q...
44  00 00 00 64 74 00 05 68 65 6C 6C 6F                   ...dt..hello
```

Fig. 7.2: Hex view of the serialized object $o1$

we will take a closer look a the serialized format of the object and see what each byte represents.

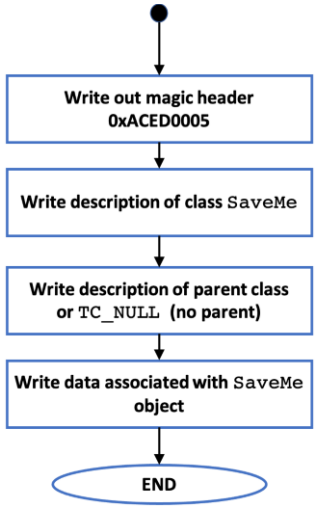


Fig. 7.3: Outline of the Serialization steps for class *SaveMe*

7.3 Java Object Serialization Protocol Revealed

As already discussed, Java's object Serialization creates a binary stream. Unlike JavaBeans persistence, for example, it is not readable by humans. Fortunately, the format is well documented. Oracle provides corresponding documentation on its website in which details of the *Object Serialization Stream Protocol* are presented [56]. The specification defines context-free grammar for the stream format. It gives a good insight into the Serialization process. The stream rules formulated in it are used directly in the Serialization of an object. In addition, a look at the source code of the *ObjectOutputStream* class reveals a lot about concrete implementation. Fig. 7.4 shows the first part of the grammar using a syntax diagram. It defines a set of

production rules (<R_n>). These rules can then be used directly to generate an object data stream.

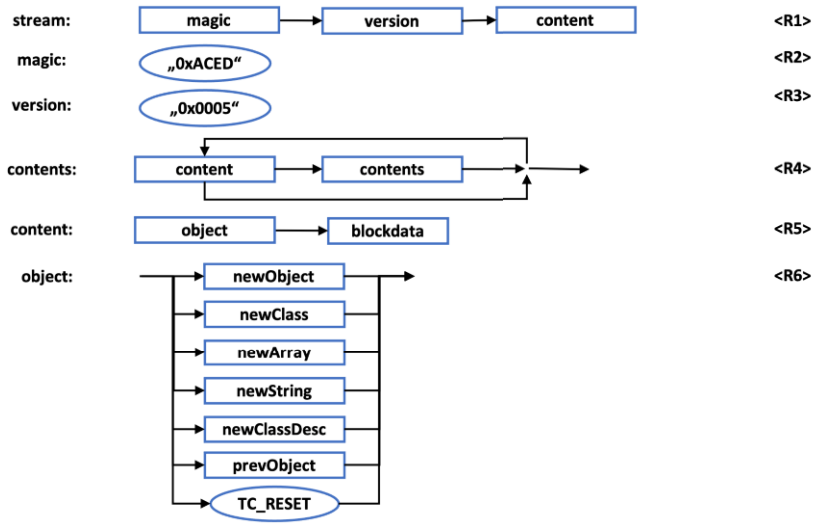


Fig. 7.4: Syntax-diagram for the Java Object Serialization Protocol (Header Detail)

The syntax diagram for this is thus as follows: Definitions of symbols are followed by a ":". We first distinguish between the terminal and non-terminal symbols. The latter can be recognised by the fact that they consist of a literal enclosed in double quotes. Constant values are enclosed in an oval. A rectangle marks non-terminals. A sequence of values is represented as a series of symbols on the same line. The individual values can be found precisely in this order in the stream. A definition consists of zero or more alternative values. Alternatives are indicated by a branch (exclusive OR).

Let us now turn to the concrete meaning of the rules. Each object stream initially consists of a magic number, the version number and the actual content specification (R_1). A magic number opens the data stream. The 2-byte integer value resides on offset 0. On offset two, the stream version field follows. According to the internal specification, this is assigned to value *0x0005* (R_3). Thus, a Java object stream can be detected with the help of the header signature 0XACED0005 (R_2 + R_3). This information is beneficial when carving to serial format files on disk. The actual content directly follows the header in the data stream.

The *contents* field is defined recursively (see R_4). Thus it can hold multiple content objects. A *content* object is first divided into an object description and a data block with the concrete attribute values (R_5). Thus, valid values for a content element are *objects, classes, arrays, strings, enumerations, exceptions* (R_6). In this way, all elements of a class and its objects can be described. Within the byte stream,

limiter symbols indicate the type, start and end of particular elements. These terminal constants (TC) are shown in Table 7.1.

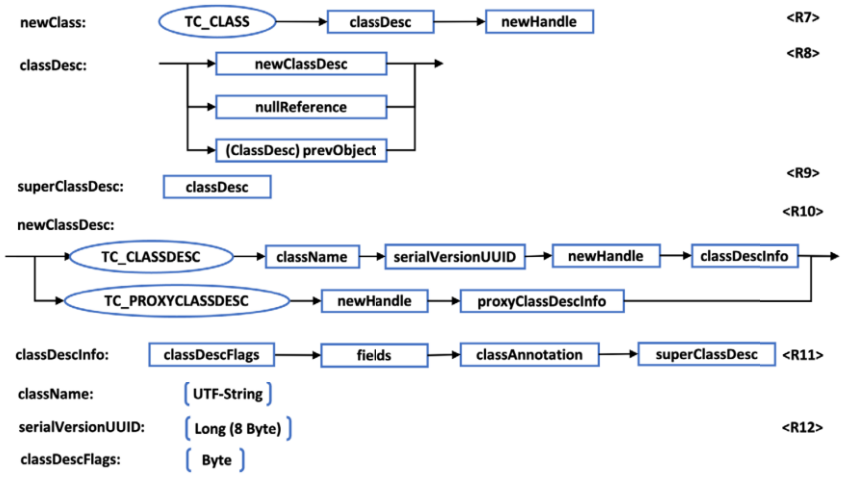


Fig. 7.5: Syntax-diagram for the 'newClass' production rule

Table 7.1: Stream Terminal Constants (TC)

Constant	Value(hex)
TC_NULL	0x70
TC_REFERENCE	0x71
TC_CLASSDESC	0x72
TC_OBJECT	0x73
TC_STRING	0x74
TC_ARRAY	0x75
TC_CLASS	0x76
TC_BLOCKDATA	0x77
TC_ENDBLOCKDATA	0x78
TC_RESET	0x79
TC_BLOCKDATALONG	0x7A
TC_EXCEPTION	0x7B
TC_LONGSTRING	0x7C
TC_PROXYCLASSDESC	0x7D
TC_ENUM	0x7E
baseWireHandle	0x7E0000

Fig. 7.5 describes the syntax definition for a class. Besides the class name, the *SerializationUUID* and *ClassDescInfo* elements are of particular importance (see R_{10}). Note: A corresponding class description must first be placed in the data stream

for each object to be Serialized. It contains information about attribute names and data types. If the class has been derived from a special super-class, the class must, of course, also be Serialized. Since attributes of the super-class are also inherited in the deriving class, we must also capture them. If the super-class, in turn, has a parent class, then this must also be described. Inheritance relationships thus significantly increase the data stream. Fortunately, Java supports only single inheritance.

Particular attention should be paid to the symbol TC_CLASSDEC. It is used to show the start of a new class definition. The byte TC_OBJECT (0x73) represents the start of an object. A data block (TC_BLOCKDATA) is in turn initiated by the byte value 0x77. We need to search the serial stream for these symbols to make the data visible. The constant *baseWireHandle* is of particular importance. Each Serialized element is assigned to such a handle. The first Serialized element contains the handle 0x7E0000, the next object is the trade 0x7E00001 and so on. In this way, for example, an object can reference its class.

Table 7.2: Type Codes / Stream Symbols for Primitive Types

Symbol	Datatype
B	byte
C	char
D	double
F	float
I	integer
J	long
S	short
Z	boolean

Table 7.2 shows the identifiers used in the stream for the eight built-in data types in Java. We can now start decoding the first part of our sample file <saved.ser> with what we have discussed so far (see below). As discussed above, the binary stream is opened by the magic number (0xACED) and the stream version (0x0005). The next byte indicates that this is an object that follows (0x73). The following byte introduces the class identifier (0x72).

> Important

Java serialized objects have a specific signature. We can use it to identify an object stream. The binary value is 0xACED0005. It translates to BASE64 as “rO0ABQ==” in a HTTP-Stream for example.

The sub-element is composed of the className. The class name is again composed of a length specification 0x0006 and the actual name string 0x536176654D65.

```
   |0x00|ACED0005 73720006 53617665 4D650000 |....sr..SaveMe..|
   |0x10|00000000 00010200 03440001 64490001 |.........D..dI..|
   |0x20|784C0001 73740012 4C6A6176 612F6C61 |xL..st..Ljava/la|
   |0x30|6E672F53 7472696E 673B7870 40091EB8 |ng/String;xp@...|
   |0x40|51EB851F 00000064 74000568 656C6C6F |Q......dt..hello|

STREAM_MAGIC - 0xACED
STREAM_VERSION - 0x0005
Contents
  TC_OBJECT - 0x73
    TC_CLASSDESC - 0x72
      className
        Length - 6 - 0x0006
        Value - SaveMe - 0x536176654d65
      serialVersionUID - 0x0000000000000001
      newHandle 0x007E0000
      classDescFlags - 0x02 - SC_SERIALIZABLE
      fieldCount - 3 - 0x0003
      Fields
        0:  Double - D - 0x44
            field name 'x'
            Length - 1 - 0x00 01
            Value - d - 0x64
        1:  Int - I - 0x49
            field name 'd'
            Length - 1 - 0x00 01
            Value - x - 0x78
        2:  Object - L - 0x4C
            field name 's'
            Length - 1 - 0x00 01
            Value - s - 0x73
            class name
            TC_STRING - 0x74
              newHandle 0x007E0001
              Length - 18 - 0x00 12
              Value - Ljava/lang/String; -
              0x4C6A6176612F6C616E672F537472696E673B
      classAnnotations
        TC_ENDBLOCKDATA - 0x78
      superClassDesc
        TC_NULL - 0x70. <- end of class description
```

integer. The value in the example is 0x0000000000000001. The *SaveMe* class is internally assigned with the handle 0x007e0000. The handle never appears directly in the stream. Only if later, a stream element again refers to the class will be visible in the stream. Various flags typically follow the class name. This flag indicates that this class supports serialization (0x02). Now we have to read out the actual attribute values. They follow directly after the class description. The object we stored in the above example has a total of three more attributes: 'x', 'd' and 's'. Since the serial number attribute is not counted, the property *fieldcount* has the value 0x0003. The individual attribute descriptions with a data type, name and length specification

follow directly afterwards (see Fig. 7.6). The string attribute is different here. Since this value itself is an object and not a primitive data type, it is also handled. Again, this is not displayed in binary code. However, Java carries an internal counter. The TC_ENDBLOCKDATA (0x78) value marks the end of the class description. Next, the serialization algorithm checks to see if the current class has any parent classes. If it did, the algorithm would start writing that class. Since we have not specified a superclass, the superClassDesc field remains empty or is assigned the terminal symbol TC_NULL(0x70). Finally, the actual attribute values for our object *o1* are still missing:

```
|0x30|6E672F53 7472696E 673B7870 40091EB8 |ng/String;xp@...|
|0x40|51EB851F 00000064 74000568 656C6C6F |Q......dt..hello|

 newHandle 0x00 7e 00 02
    classdata
      SaveMe
        values
          'd' (double)3.14 - 0x40091EB851EB851F
          'x' (int)100 - 0x00000064
          's' (object) TC_STRING - 0x74
                newHandle 0x007E0003
                Length - 5 - 0x0005
                Value - hello - 0x68656C6C6F
```

As we surely noticed, the object name *o*1 is missing. This information is not stored in the stream since it is only an identifier. The programmer decides under which identifier the object can be accessed after deserialization. The value assignment of the ordinal types *d* and *x* follows directly in the stream. The first attribute has 8 bytes in length for the LONG value. The integer value has a length of 4 bytes. Thus, unlike database formats such as SQLite, no compression is used. The memory content is transferred 1:1 into the stream. The string value terminates the stream. A string is not an ordinal type but an object itself. Therefore first, the identifier follows as a string (0x74). The length of a string is dynamic. Therefore the length specification is additionally prefixed to the string value (0x05). The actual value follows last. There is no particular end identifier. Instead, the stream ends [54].

There are some production rules which are not listed so far. Special stream types like enumerations, exceptions or proxy classes are missing. At this point we refer to the protocol description on the oracle website [57][56].

7.4 Pitfalls and Security Issues

In the last years, the serialization protocol of Java was increasingly in the criticism due to different vulnerabilities [76][34]. At this point, we want to shed light on the background and show how these threats can be minimized.

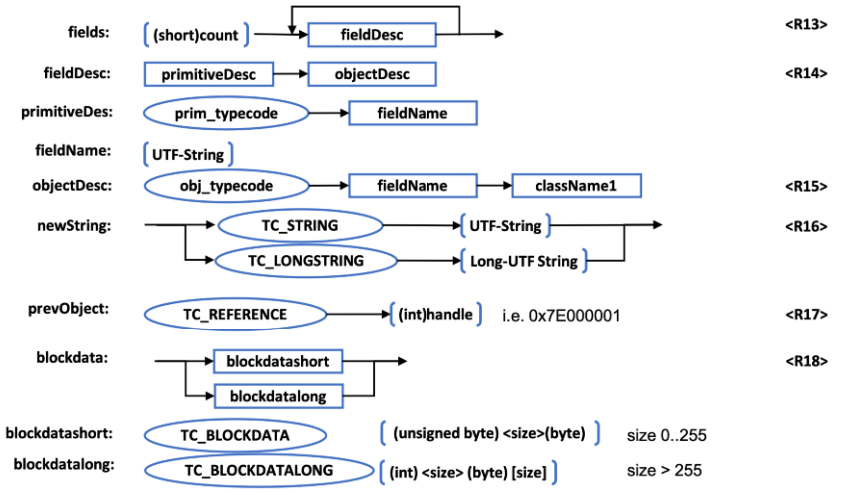


Fig. 7.6: Syntax-diagram for the objects 'fields'and 'blockdata'

7.4.1 Hands on Serialized Objects

With what we already know about serialization, it is easy to find a point of attack. It is not uncommon, although it is not recommended, to transmit or store important confidential information in streams. Assume a user-supplied serialized object we discovered. We can manipulate program logic by tampering with the information stored within the stream. For example, let's take our object *SaveMe*. We can easily modify the string at the end of the stream:

```
0005 68656C6F | ..hello    ->  0007 62656576 696C  | ..beevil
```

Since there are no parity bits or checksums, we do not need to adjust anything else within the stream. In our case, that is probably not a big deal. If somebody cheats on a gaming app and overwrite the high score, we can certainly get over it. However, the whole thing changes quickly when confidential data is included in our stream (e.g. user=admin, password=abc123). For example, if the Java object is used as a cookie for access control, we can change the usernames, role names, and ID-token. One can also try tampering with any value in the object that is a file path. We can even alter the program's flow if we override the correct field.

7.4.2 Beware of Gadget Chains

As if that were not enough, we can sometimes even perform remote code execution (RCE) [34][76]. In Java applications, so-called *gadget classes* can be found in the libraries loaded by the application. Using gadgets that are in-scope of the application,

we can create a chain of method invocations that eventually lead to RCE. This chain can be bumped during or after the deserialization process.

Listing 7.4: Possible Vulnerable Class

```
public class Vulnerable{
    public Object invoke(SaveMe o) {
        return Runtime.exec("echo \"I just want to say\"" + o.s);
    }
}
```

An example class is shown in the listing above. The *invoke()* method in this example uses the string attribute we modified earlier. If we set the string *s* to "hello | mkdir somedirectory", the second part of the statement causes the creation of a new directory on the target system. This sort of attack is called *Command injection*. Alternatively, we can, of course, execute any command we like. The goal is to execute arbitrary commands on the host operating system via a vulnerable process. Web servers are very prone to this form of attack.

All we need to find is an appropriate hooking point. Therefore, we should look for gadgets in commonly available libraries to maximize the chances that this gadget is in-scope of the application. To date, exploits utilizing gadgets are already known and published. Those classes are mostly part of popular libraries such as the Apache Commons-Fileupload, Apache Commons-Collections, or Groovy. A collection with the gadget chains for Java can be found in the *ysoserial* project from Chris Frohoff [3]. The repository offers a collection of utilities and gadget chains discovered in shared Java libraries. Due to unsafe serialization, a gadget chain may automatically be invoked and cause the command to be executed on the host system. The creation of an unsafe serial object with *ysoserial* is straightforward:

```
$ java -jar ysoserial.jar [gadget chain] '[command to execute]'
```

The dangerous thing about this is that it does not depend on what classes we use in our application. It is sufficient that the class in question is accessible via the local classpath. To honour the rescue of Java developers, however, it must be said that this is not only a particular problem of Java Runtime Environment. Such security issues can also be found in languages like Python, PHP, or Ruby [92],[87].

However, how can we prevent such attacks now? One measure is to blocklist or allowlist object classes before deserializing them. Most suitable for this is the *resolve()* method of the ObjectInputStream class (see Fig. 7.7). If we would validate the object directly after *readObject()* has finished its work, it may already be too late. However, if serialization is performed by a framework class working in the background, we do not even have to notice it.

[3] https://github.com/frohoff/ysoserial

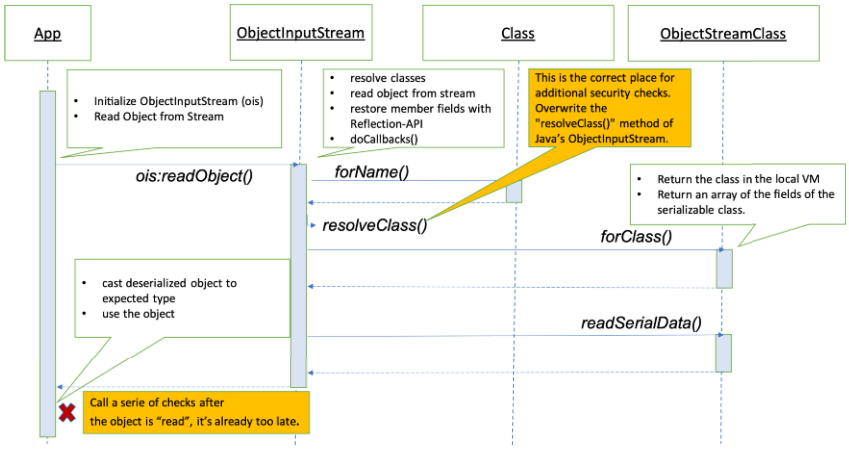


Fig. 7.7: Sequence Diagram of Object De-Serialization

7.5 Conclusions

In this chapter, insight into the standard Serialization format of Java was given. Serialized objects are used in many places in Java. Despite the security problems mentioned earlier and the relatively modest performance, the format enjoys unbroken popularity. Forensically, the file format is exciting because it is not uncommon for confidential or sensitive data to be stored in a stream. However, it should not be a problem to restore attribute assignments with the appropriate tools, even for unknown classes.

Chapter 8
Realm

Phil Cobley and Ginger Geneste

Abstract In this chapter, we explore some of the fundamentals of the Realm database (sometimes referred to as RealmDB or simply Realm). It is widely known within the Digital Forensics discipline that SQLite is the most commonly found database format within any mobile device application and even some desktop applications. Realm is a relatively new database format built as a potential replacement for SQLite, as technology and applications continue to develop and evolve. At the time of writing, it is clear that the database is not as commonly found as some might have expected, but that is not to say the database format will not eventually find its way into many modern apps over the coming years. To that end, we decided to research the database to try and provide some of the details of interest relating to the fundamentals behind the new format. We hope this chapter will help digital forensic examiners and investigators learn and grasp some of the basic concepts of Realm, hoping that any new knowledge and understanding might support and assist in future research into the topic.

8.1 Organisation of this Chapter

You will find a chapter dedicated to the SQLite file format within this book. We shall be covering some of the basics behind SQLite for ease of readability and not assume prior knowledge. We look at some of the differences between SQLite and Realm before looking at Realm in more depth. We want to highlight that the Realm code is under constant development and is still in the early stages of that development,

Phil Cobley
MSAB, 2nd Floor East, Central Point, 25-31 London Street, Reading, RG1 4PS, UK e-mail: phil.cobley@msab.com

Ginger Geneste
Netherlands Forensic Institute, Laan van Ypenburg 6, 2497 GB Den Haag e-mail: g.geneste@nfi.nl

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_8

meaning there are some limitations as to what can be confirmed both in the short and long term.

We have broken the chapter down into sections to help readers navigate through a journey of discovery, starting with some high-level and generic concepts, eventually drilling down into more detail later on. Each section builds on the last, but we have tried to write this chapter so that you can easily use the content as reference material if needed.

We start by looking at some of the similarities and differences with SQLite, exploring the concept of object-oriented database design and development over traditional relational tables and SQL queries. We then move into looking at how Realm works and how data is structured similar to a table-like format, but without actually creating any tables. We detail some object-oriented concepts before exploring the Realm files and their data structures in far greater detail. We discuss the concept of data and reference arrays and how these play an important part in Realm databases and break down the various file and array headers at the byte level.

We use example files in the chapter that can be created or downloaded, with the links available within the text itself, if you wish to either create those files or download them yourself. We use those files to break down a Realm array, looking at the offset pointers, the header, and how we can examine the data to identify the size of the payload and the type of data each array contains.

8.2 Introduction

In Digital Forensics, we are often interested in collecting and reviewing data held in databases. Most applications and operating systems rely on databases to store, organise and manage their data instead of swathes of unconnected files and unsorted data blocks. Databases make storage and retrieval simple and provide standardised mechanisms and schemas that modern applications can easily harness.

Within mobile device forensics the most commonly found database type used by applications is SQLite [78], which is a cross-platform, serverless database type, that has become a valuable tool for mobile app and operating system (OS) developers over recent years. The SQLite database was designed to be simple to use, easy to connect to applications across any platform, and could be installed and run upon the client device without the need for a bulky, backend server [83].

However, as devices, applications, and our usage needs of mobile devices evolve, so do the databases harnessed by applications. SQLite is a powerful database format, but it has its limitations. Often, modern-day developers are forced to generate and write additional code to enable their applications to do what they need them to do due to the emerging limitations present within SQLite. This additional code often involves implementing workarounds to enable natively unsupported data values stored within the SQLite tables.

This chapter seeks to explore the Realm database format [71], which has emerged over recent years as a possible successor to the now ageing SQLite format, looking

at how this database format plugs those emerging gaps. Understanding how this database structure differs from SQLite should enable forensic examiners to understand better the types of data they are likely to encounter and appreciate how the format works in practice. For example, Realm databases do not use relational tables, a core feature within SQLite databases, but instead, work with linked objects. How does this impact forensic analysis? In this chapter, we shall look at what artefacts we can expect to find when a Realm database has been used and clarify what data may be found within.

! Attention

It is worth noting that research into this subject is still ongoing, and so, while this chapter seeks to explore how Realm databases work, it is not a comprehensive deep-dive into the full workings and data structures. This chapter may expand and evolve in future revisions of this book. However, we have ultimately tried to incorporate as many confirmed findings and factual content as possible at the time of writing. You will see that we have included enough for researchers to understand the fundamentals of these data structures and for forensic examiners to decode various headers and attributes, and we hope that this is the strong starting point to encourage and support examiners in taking this research further.

8.3 SQLite, It is Not!

While Realm might be replacing SQLite in some applications, the way they are coded and operate differ greatly. In order to understand how the object-oriented approach of the Realm database structure works, we will first go through an introduction to the more common relational database structure. We will then explore the concepts of an object-oriented approach for database structures, comparing its features with that of a relational database.

8.3.1 Relational Databases

There are many ways to define what a relational database is. However, in essence, it is a method of organising data into tables, which are linked together through common criteria or data components [38]. Data is typically organised into rows and columns, with each row being assigned a unique identifier. Tables can then reference data in other tables through the use of these unique identifiers, which is the "relational" aspect of the relational database concept.

As a simple example of how this might work, imagine we wish to use this concept when looking at grocery shopping. When grocery shopping, we may wish to search

for the items we need by searching under specific categories, such as fruit, vegetables, meats, bakery, and so on. In a relational database setup, it may be that these are each represented as different tables, each containing the various items that you might find under that category (see Table 8.1).

Table 8.1: Example Grocery Tables

Fruit	**Price**
Apple	0.2
Orange	0.3
Pear	0.15
Banana	0.25

Vegetables	**Price**
Carrots	0.11
Potatoes	0.15
Cabbage	0.2
Cauliflower	0.3

Bakery	**Price**
Bread	0.8
Rolls	0.4
Wraps	0.95
Bagels	1.6

Structured Query Language (SQL) [39] is often used as a standardised language to both write data to and query data from such databases that support it. It has enabled developers to quickly and easily write and format queries and code to edit and pull data from relational databases through a global standard. Tables are connected to one another through functions known as “Joins” with search queries generating combined results sets in newly created tables containing various record (row) content, depending on the query made.

When carrying out searches across data sets, such as those found in our grocery example, SQL may be used in the background of a website or application to run queries across the tables, utilising search terms input by the user. This may include filters we commonly see on websites to narrow down the search. For example, we may be looking for a loaf of bread and therefore click on a "Bakery" filter and search for the term “bread” Fig. 8.1. SQL may then be used in the background to search for row items containing the keyword “bread”, but only within the “Bakery” table. Equally, applying no filters may conduct the search across all tables, thus conducting a wider search.

When searching such as this, there may also be an empty table either created or already available, that is used to hold copies of the search results. The column structure would likely be very similar to have compatibility with the existing data sets from the other tables but may have additional columns specific to a search. You could think of this table as possible where a typical search results page on a website may be drawing data from.

There are obviously countless ways to develop and programme these structures, and so this is just one (very simplistic) possible example of how data may be held, linked, and manipulated within a relational database. However, you will often have tables of fixed data content that are used as a reference point for the application. You will also have other tables populated and edited through user interaction or system processes, containing live or deleted content.

If we use our grocery example from a forensic standpoint, it might be that the grocery item tables are of little interest to us. However, the search table, or possibly

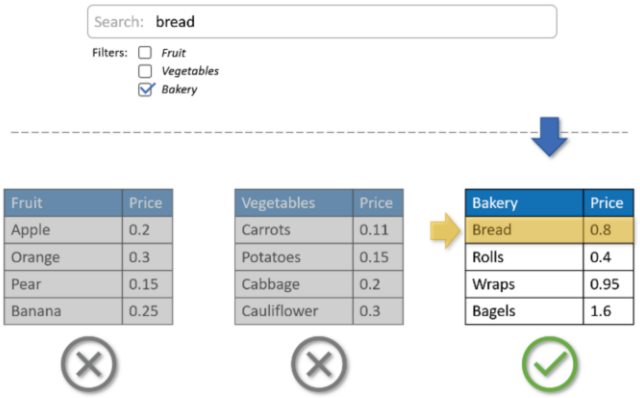


Fig. 8.1: Background Table Searches

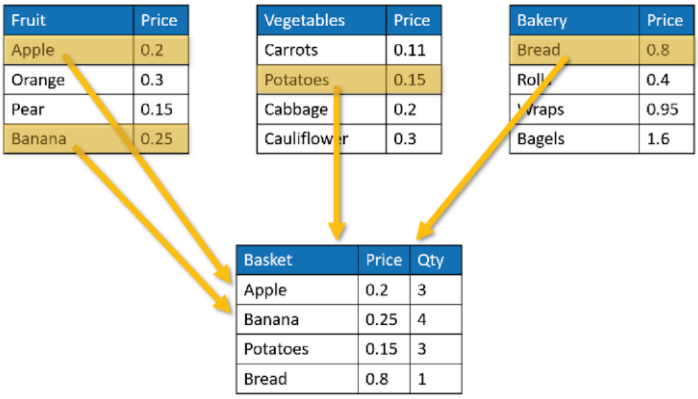


Fig. 8.2: Relational Table Results Collated in a New Table

a "Basket" table that is populated by the user when they add items to their basket might be of forensic interest as it would help identify user activity.

8.3.2 SQLite as a Relational Database

SQLite uses a relational database structure, storing all of the data tables and links within a single file, usually with a file extension similar to *.sqlite or *.db, although this can vary depending on the intended software platform and how the developers decide to build their applications (in mobile devices you will often find them with no file extension at all). This use of a simple, single, self-managing file, is what has helped make SQLite so popular with mobile developers, as there is no need to

rely on additional backend servers, and the data becomes self-contained and very portable. These days we see mobile devices with huge storage capacities, but back when smartphones were first coming onto the market storage was still at a premium. SQLite databases enabled developers to build database storage into their mobile applications without taking up very much space on a user's device, and without the need to install additional software or depend on addition software or services running in the background, taking up valuable processing capacity, memory, or network bandwidth.

SQLite is also a cross-platform file format, which means it can be run and used on any common or major operating system, reducing the need for developers to concern themselves with huge changes in their application architecture when developing for multiple platforms, such as Android and iOS.

Given that SQLite utilises SQL as a query language, and given both SQL and SQLite are platform agnostic, it means that a software developer building an application in Swift for iOS will most likely utilise the same or similar SQL queries to a software developer building the same application in Java or Kotlin for the Android operating system. This, in turn, means that the structure, layout, and logic of the backend database does not have to change very much from one system to the next, allowing app developers to focus on only having to adapt the code that surrounds the database when building for different systems, rather than having to also be concerned with what database to use and how to update, edit, and retrieve data from within it.

In digital forensics this is great news for examiners and investigators, as most of the time we only need to concern ourselves with the content of the database – and for a standardised database format such as SQLite that is found on both iOS and Android, it means we only need to learn how to interrogate one database format, regardless of what type of device that data resides within.

8.3.3 SQLite Schema

A schema is essentially a specification confirming and defining the structure of a database, usually written or presented within the appropriate language or format for that database type. SQL developers define their database schema using what is known as Data Definition Language [77] which is used to create tables and define the types of data that each column should hold, such as integers, dates in specific formats, strings, and so on, as well as stating what columns can or cannot contain NULL values (see Fig. 8.3).

8.3.4 Temporary SQLite Files

Something that we commonly find alongside SQLite databases are the shared-memory (SHM) and write-ahead log (WAL) files, that we may find located in the

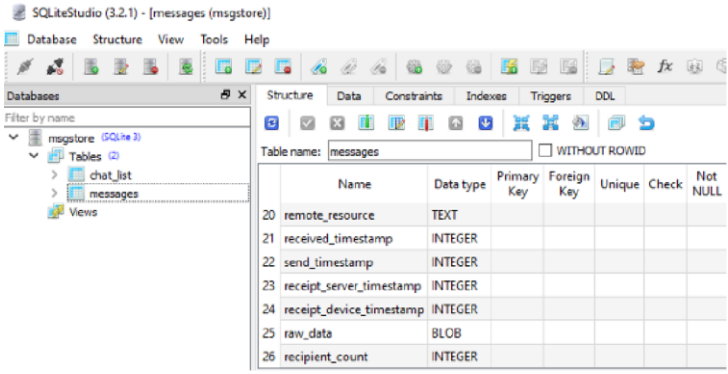


Fig. 8.3: SQLite Studio Displaying SQL Schema

same directory location as the SQLite database file [86]. There are actually several more additional temporary files that are sometimes found as well, but we will not go into detail of how these work here as they are explained in more depth within the SQLite chapter. However, we will touch on what the commonly found WAL and SHM files do and why they (sometimes) exist, for our later comparison.

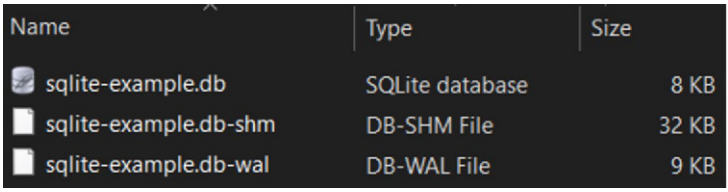


Fig. 8.4: SQLite Files, including SHM and WAL

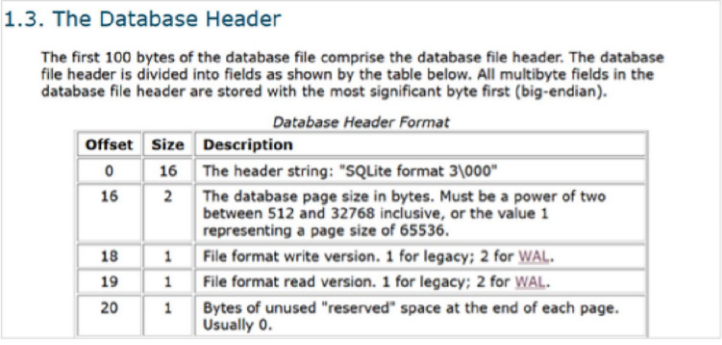
The WAL file is a form of journal that stores updates and changes to the SQLite database file. Rather than writing changes directly to the database file itself, which could cause complications and problems in numerous possible implementations of the database, those changes are written to the WAL first. Multiple changes and amendments can be made (resulting in duplicate entries within the WAL), pending the appropriate trigger to "Commit" those changes in the WAL. This committed content only gets written to the database itself once a "checkpoint" is triggered, where the most recent versions of all the changes and amendments are then transferred across to the database itself. Those triggers vary depending on the version and implementation of SQLite, but occasionally the commit does not take place during a single session, and may only be triggered when the database is reopened and accessed later on. This means that the WAL file can persist in a file system even after a connected application is closed, and it is not actually uncommon to find the WAL file to be even larger in size than the database itself.

The SHM is a file created to help manage concurrent connections to the database and allows the WAL to use a specified area of memory for indexing and managing the various changes and commits being made to the database. In essence, if you have multiple system services or process threads access the SQLite database file at the same time (which is very common) then an SHM file will be created to help service those connections. If one is created then it will typically persist on disk with the WAL file until the WAL file is deleted.

Why are these files important? Well, it is not uncommon for forensic analysists to find evidence and vital data buried within these temporary files, rather than the data being within the main database file itself. We shall see how this differs within Realm databases later on.

8.3.5 SQLite File Format

SQLite is an open-source file format with a very distinct and well documented structure. A lot of digital forensic training and education programmes will teach examiners about the specific layout and structure of the SQLite database header, as understanding the various attributes and byte values can be invaluable, particularly when dealing with more complex and challenging forensic scenarios where the database cannot be automatically parsed and decoded, albeit these instances are rare due to the wide range of comprehensive forensic software tools available. The header information can be found at: https://www.sqlite.org/fileformat.html, (see Fig. 8.5) [82]).



1.3. The Database Header

The first 100 bytes of the database file comprise the database file header. The database file header is divided into fields as shown by the table below. All multibyte fields in the database file header are stored with the most significant byte first (big-endian).

Database Header Format

Offset	Size	Description
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for WAL.
19	1	File format read version. 1 for legacy; 2 for WAL.
20	1	Bytes of unused "reserved" space at the end of each page. Usually 0.

Fig. 8.5: Screenshot from sqlite.org File Format Webpage

You can visit the referenced website to find the full database header format content, along with explanations and documentation around each offset specification.

8.4 How Realm Works

8.4.1 Realm Database Fundamentals

Realm is described on the realm.io website as:

> *"... an open source, developer-friendly alternative to CoreData and SQLite. Start in minutes, port your app in hours, and save yourself weeks of work."* [71]

The database itself is an object database as opposed to a relational database, harnessing the principles of object-oriented programming over traditional database models such as SQLite. This approach allows the database to benefit in ways that are not possible in SQLite, such as having a zero-copy architecture and a near endless possibility to handle, store, and manipulate almost any file format or data type with ease.

As we already know, relational databases, such as SQLite, consist of tables that join and work together to reference various data sets, locating and identifying records and data by navigating rows and columns. Sometimes the table connections (joins) can become incredibly complex, often requiring tables built specifically to hold unique reference variables to help tables navigate to and reference one another. Any queries that are run have their results copied into another table, duplicating data content for the purposes of generating query results. This can be time consuming, memory and processor intensive, and take up considerable data storage as databases grow and expand. The sheer act of copying data out of tables to represent the same content within another table could also be seen as being inefficient and unnecessary.

In their paper "Evolution of Object-Oriented Database Systems" [2] Alzahran compares traditional relational data models with object-oriented models, discussing the future of database structures and a shift in the current paradigm. An example of an object-oriented model (such as Realm) and relational data model (such as SQLite) is given in Fig. 8.6.

In the example presented in Fig. 8.6, we can see how data tables in a relational model are replaced with object instances of different classes, with a new object instance being created instead of a new row being added to a table. The columns in a table are now represented through object attributes, meaning that in order to locate data the object instance is queried and asked to return the attribute values, rather than tables being queried through SQL expressions.

8.4.2 Common Concepts and Terminology

Here we shall define and provide an overview for some common concepts and terminology used within Realm database architecture.

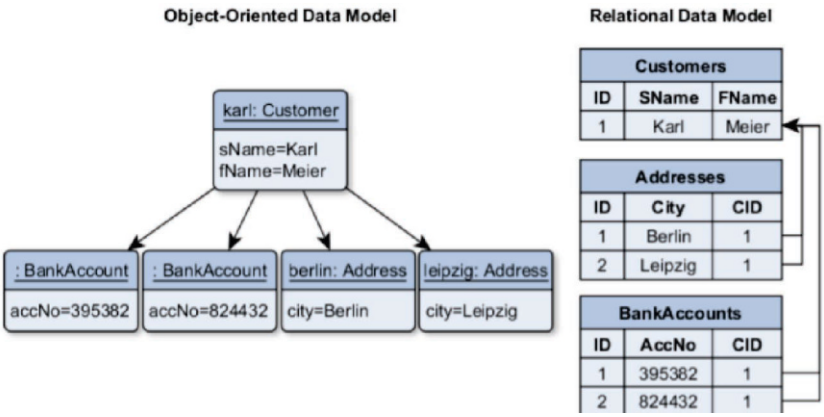


Fig. 8.6: Screenshot from sqlite.org File Format Webpage

Basic Object-Oriented Programming Concepts

In object-oriented programming, used in languages such as Java and Python, there are a number of concepts and principles that are considered as very important when it comes to software design and development [16]. One of these concepts is known as "low coupling, high cohesion" and another being the concept of having small, well-defined objects that do specific jobs, rather than large, bulky objects that carry out multiple tasks.

The reason behind these concepts helps promote code design that is flexible and dynamic as well as being easily maintainable and adaptable. By having small, single function objects, you can build code where any other objects requiring that function simply call those specific object instances, rather than running the risk of duplicating the function within many larger objects. While singular, self-sufficient objects may seem like a good idea, it makes the code more difficult to maintain and manage. For example, say a specific function requires updating; if it is duplicated within multiple object types then they all require updating to ensure no objects are running with legacy code. However, if you have a single object for that function that other objects call upon to use, then simply updating that one object function will update the capability for all connected objects with minimal updates being required. This concept is sometimes referred to in other industries as "single-source" and relates to many different practices, not just software development.

The concept of low-coupling and high cohesion links into this through design principles in software development that suggest objects should work well together (high cohesion) but remain independent so that changes to one do not negatively impact the other (low-coupling). This allows developers to design code that is more easily maintainable and resilient to change. If objects can work well together and have a means to communicate without being dependant on exactly "how" the code

has been implemented, then when that code needs changing or updating, so long as the communication methods remain in place, other objects remain unaffected.

Realm databases are able to leverage these benefits through an object-oriented approach to the database design, where the application creates and maintains lightweight, connected object instances as opposed to bulky, rigid, relational table structures. In SQLite, complex systems and database designs often rely upon queries that fully understand and recognise exactly what they're looking for, where, and how. The benefit this brings forensic examiners is that those queries and table structures are relatively straight-forward to reverse engineer and piece together, given how explicit the calls and queries often have to be. In Realm, this is not necessarily the case, as the database queries and calls are highly dependent upon how the developer has decided to implement the Realm database objects and instances, how they have coded the various communication methods and object attributes, and how complex the communication structure ends up becoming. The way this often happens is through objects communicating in a chain, from one to the next, making singular queries to one another, rather than a single, large, complex query statement across multiple tables (Fig. 8.7).

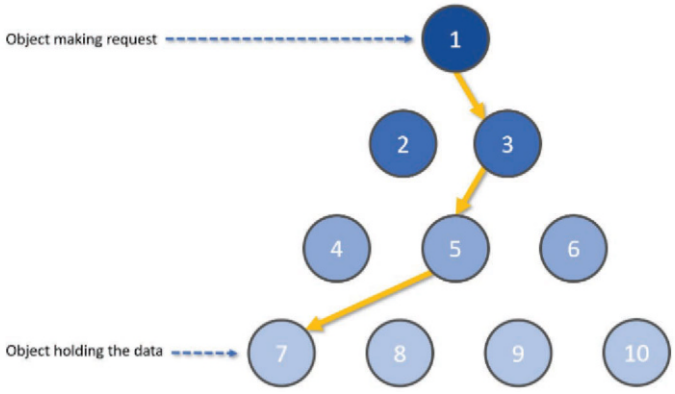


Fig. 8.7: Object Instance Communication

Top-level Objects

Top-level objects could be considered the equivalent of a relational table within an SQLite database. They are typically found to be object classes, such as "Fruit" or "Vegetables", similar to our example earlier in section 8.3.1.

In the following example we use a screenshot from a demo file available for download from the realm.io website [72] viewed within the official free Realm Studio tool [74], used for testing and training purposes to demonstrate some of what Realm can do.

In the screenshot you can see:

- Listed object classes – these are equivalent to a relational table
- A list of object instances – these are equivalent to the rows/records in a table
- The defined object properties/attributes – these are equivalent to table columns

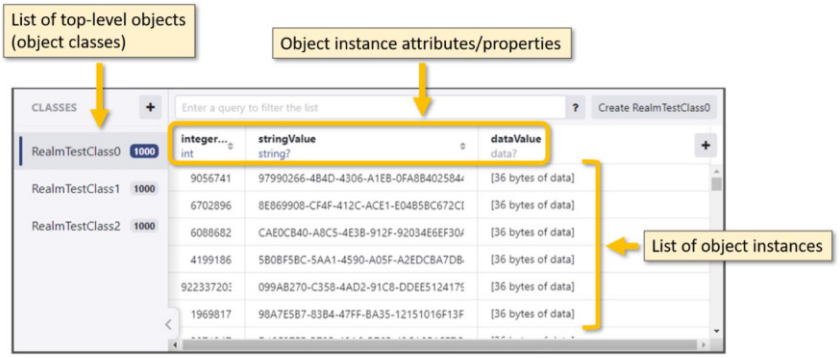


Fig. 8.8: Realm Top-level Object Structure Example

Object Types

It may sound obvious to some, but within Realm databases, an Object Type is a term used to help define exactly what any given object is recognised to be by the database. Object Types link to the database schema, which is defined in code by the database developer and works in a similar way to that of a database schema within SQLite. However, this is one area where Realm goes beyond the capabilities of SQLite in its capabilities.

In SQLite schemas column values are restricted to predetermined values, such as string or integer values, with a finite list of available types that are coded into the SQLite codebase. Anything that does not fit within this predefined list is typically managed through the use of BLOB data, which stands for Binary Large Object [49]. BLOB data can be almost anything that a developer wishes to include, but the data management can sometimes be complicated, and sometimes requires encoding. Furthermore, the database itself will usually not be able to determine exactly what the BLOB data represents, as this has to sometimes be managed through additional software components or tools outside of the database environment.

Within Realm , as the entire schema can be developed alongside the object code, the “type” values can be absolutely anything that the developer wishes to include or use, so long as the type represents a coded object class. This means that proprietary data objects can be built within the Realm code, teaching the database how to handle,

manage, store, and manipulate that data natively, regardless of what it is or how it is constructed.

This can be potentially very powerful for database and application developers, but prove to be a huge challenge to forensic examiners and investigators, as it may require reverse engineering the original code in order to understand any proprietary or custom object types or formats. The MongoDB documentation for Realm has an example schema which shows possible schema code for storing data about books in libraries, where "Library" and "Book" are object types [49]:

```
[
 {
    "type": "Library",
    "properties": {
        "address": "string",
        "books": "Book[]"
    }
 },
 {
    "type": "Book",
    "primaryKey": "isbn",
    "properties": {
        "isbn": "string",
        "title": "string",
        "author": "string",
        "numberOwned": { "type": "int?", "default": 0 },
        "numberLoaned": { "type": "int?", "default": 0 }
    }
 }
]
```

Every object within a Realm database must be of a type that is validated by the schema and properly defined. It is worth noting, given Realm is based on the concept of everything being an object instance, that the Realm database itself is an object of type "realm". When a Realm database file is opened and accessed an instance of the Realm database is initialised, with the relevant attributes and properties loaded into that instance from the stored data.

Group

When a Realm database is accessed or opened, the schema is read and interpreted to begin validating and initialising the appropriate object instances. Groups are collections of top-level objects (so, the equivalent of a collection of tables in SQLite) which together help identify and clarify the schema requirements.

Essentially, whenever a Realm database file is accessed through one of the Realm database SDK's, the file is verified and loaded into a Realm object by calling the Realm Group [73].

Arrays

Realm databases predominantly store their data within data arrays, and so first we shall take a quick look at what an array actually is.

It is probably fair to say that almost every programming language can implement arrays in some form, and whilst their implementation may differ depending on the language, the concepts behind them remain fundamentally unchanged. They are simply a data structure that can be used to store an ordered collection of data within a single programmable component. What does that mean, exactly? Let us use an example to explore the answer to that question. Imagine you are programming a simple application and have decided you want to assign variables to hold the names of people who are attending an event. Now, there would be hundreds of different way to do this, and some more efficient than others, but this is just a simplified example to help with our understanding.

You may decide to programme individual variables, maybe something similar to the following:

```
attendee1 = "John"
attendee2 = "Sarah"
attendee3 = "Sam"
```

This might work really well for the first few attendees, but when you expand your system to a thousand, it may begin to get tedious and time consuming, not to mention a huge amount of code. Instead, you may consider an array. This component allows you to define a certain number of elements which are automatically numbered sequentially as the values are added to the component, so it is similar to a table that has two columns and a finite number of rows. The first column is automatically determined by the array and grows sequentially from 0 upwards, and the second column is for the data you wish to assign to each row. This allows you to forget about needing to code variables for each user, and instead lets you simply add the names directly to the array, so the array might be conceptualised similar to Table 8.2.

Table 8.2: Conceptual Array

index	**value**
0	”John”
1	“Sarah”
2	“Sam”
...	...
1000	"Yvette"

The order of adding values is very important with an array, as the indexes are filled sequentially to ensure efficiency, so no gaps are purposefully left. Arrays make it very easy for the software to locate specific data values as the index can be used to

locate the data quickly. However, arrays do not care about the order in which they store their data, which differentiates them from other similar data structures where ordering and sorting is an important part of their function.

Another way of looking at arrays is like the chapters in a book. The chapters in a book are fixed and will not change, and the book provides a way for the reader to use the chapters to locate the information of interest to them. In this example, the book is the array and the chapters are the indexes and their data values (Fig. 8.9).

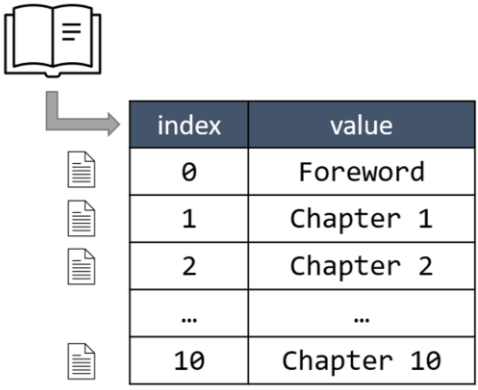


Fig. 8.9: Array Book Analogy

A Realm database file consists of a file header and is followed almost exclusively by 'Realm Arrays'. If the database is not encrypted then these arrays can be located and identified with a hex viewer and parsed out either manually or with appropriate scripting. There are two types of arrays found within Realm databases that we found through testing:

1. Arrays containing references to other arrays (referred to as a **Reference Array**)
2. Arrays containing data (referred to as a **Data Array**)

Essentially, a Realm database utilises what is known as a B+-Tree structure, where the tree can be recreated and mapped by following the pointers of the headers and reference arrays (the branches), until you reach the data arrays (the leaves). These arrays are all essentially nodes within that structure.

In the next section we shall begin exploring the structure behind some of these arrays, along with details of the Realm header and associated files that may be found with the *.realm database file.

8.5 File Storage and Structures

8.5.1 Realm Files and Folders

Here we are going to have a quick look at the files you may encounter when examining realm databases, including some of the temporary files that may be created, similar to how we sometimes find SHM and WAL files accompanying SQLite databases. Two core files will be commonly found with a Realm database, along with a folder that may be empty when recovered [48].

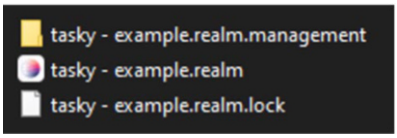


Fig. 8.10: Realm Files

! Attention

We will not be covering Realm encryption in this book chapter, but it is worth being aware that they can be encrypted at source.

8.5.2 The Realm File

The most obvious file is the Realm database itself. The Realm database is referred to in documentation simply as a "**Realm**" which is a term that encapsulates the database and all associated files and data. The Realm is a single file that has a *.realm file extension, and contains all the generated data and associated objects. Developers are encouraged to initialise the Realm instance (create a Realm file) the first time an application is opened and run on a device, which means it may be possible to have realm-based applications that have been installed, but where no database is yet present if the application has not been run since installation. Realms can be encrypted by the developers, which would mean that static analysis may not be possible through standard tools without initial decryption taking place.

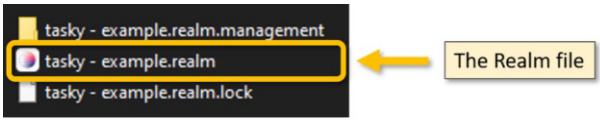


Fig. 8.11: The *realm File

The Lock File

The Lock file is created when the first connection is made, then recreated and reinitialised at the beginning of every session. This means that the file does not need to be present when the database is initially opened. The purpose of the file is to enable "synchronization between writes" [48] and even if deleted, will be recreated when the database is reopened.

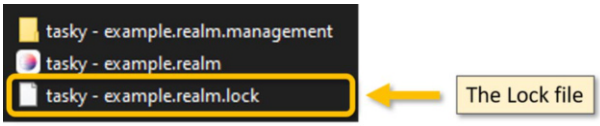


Fig. 8.12: The Lock File

A session is initiated and closed with the opening and closing of a Realm file via database objects. However, it also includes any sequence of temporarily overlapping openings of a particular Realm file via multiple database objects. For example, if there are two database objects, A and B, and the file is first opened via A, then opened via B, then closed via A, and finally closed via B, then the session stretches from the opening via A to the closing via B, rather than two individual sessions. This might be two different application instances opening the same Realm database file simultaneously, for example (like a multi-user session).

The Management Directory

This folder appears, like the lock file, when the database is opened and a connection established. Through testing we have yet to come across any files within the folder itself, but it is reported by MongoDB to "[contain] internal state management files" [48] which are likely to be of little interest within a forensic investigation.

Stateless Realm Instances

It is possible to create and run a Realm entirely within memory, resulting in no actual files being saved to persistent storage. In these instances no trace of any realm

Fig. 8.13: The Management Directory

data will be located within extracted data storage but the data may be present within extractions of any volatile memory from an active device.

8.5.3 Creating Realm Test Instance

We are going to explore two different databases through this chapter. One is a demo file provided by Realm.io, which can be downloaded directly from their website or via a link within the Realm Studio software package:

- Realm Studio can be downloaded from the following URL: https://docs.mongodb.com/realm-legacy/products/realm-studio.html
- The Realm demo file can be downloaded from the following URL: https://static.realm.io/downloads/realm-studio/demo-v20.realm

The second database we are going to look at is a simple realm database created using Java within Android Studio [22] designed as a simple tasking app.

> Important

When writing this chapter we considered including a step-by-step guide on how to create a simple Realm database within Android Studio. However, we found, through our research, that constant changes to Android Studio, Java, Realm, and associated libraries and dependencies, meant that the guides would be out of date by the time they were published, with errata being required almost immediately. We stumbled upon a well-documented guide as written by developer Joyce Echessa, published in a blog article on behalf of auth0 [12] which we have used to build the Task app referenced within this section. This has been done so that you can follow the referenced web page and create your own database, if you wish. We found we had to update a number of referenced versions and dependencies, but overall the guide was still valid at the time of writing, and we were kindly given permission to include a reference to it within this book.

Upon creating our Task application within Android Studio we now need to run our task app for the first time to initialise the database and create a Realm instance. We

then need to access our emulated device via the ADB (Android Debug Bridge) [21] to pull the newly created files out.

! Attention

If you are not familiar with ADB then then we encourage you to visit the android-studio documentation [1] to learn more about it and download the relevant software and tool packages. This walkthrough is completed using Windows 10, but you can achieve the same results on other operating systems.

Step 1: Launch the Task Application

From Android Studio, open up your emulated Android environment with your Task app present and load the operating system. Navigate to the applications list and you should see your Task app present (Fig. 8.14):

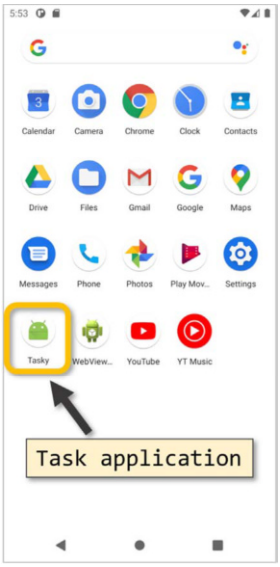


Fig. 8.14: ADB Walkthrough - Find Application

Now launch the application by clicking on the icon (Fig. 8.15) and then close it down:

[1] https://developer.android.com/studio/command-line/adb

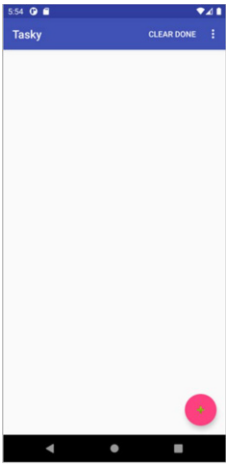


Fig. 8.15: ADB Walkthrough - Launch Task App

Step 2: Open a CMD Window

Open a CMD Command Prompt (or Powershell if you prefer), which can be done simply by opening your Start menu and typing "cmd", which will present the option to open a Command Prompt window similar to Fig. 8.16.

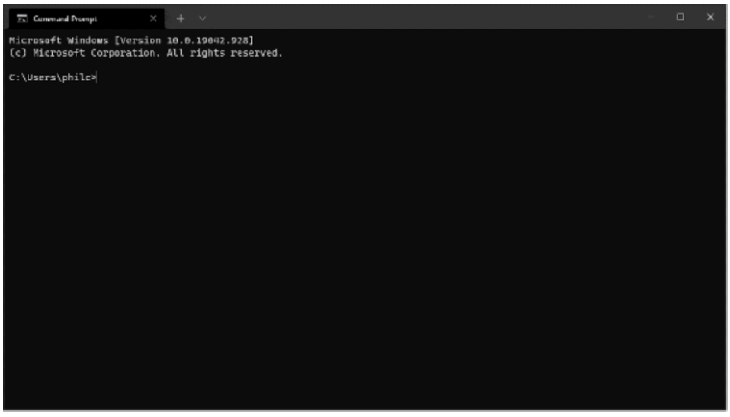


Fig. 8.16: ADB Walkthrough - Open CMD Window

Step 3: Create an Output Folder

Create an output folder where your Android files will be placed. For this example we have created a folder called “Android” at the root of the Windows `C:\` drive as this will keep the commands in later steps, much smaller and easier to manage, but you can choose any location you like.

Step 4: Start ADB

In your CMD window type the command:

```
adb devices
```

> **Important**
>
> This is assuming you have added ADB to your PATH. If not, we suggest you do this before proceeding.

What you should see is a list of devices attached to your computer via ADB (Fig. 8.17). Your emulated Android may have a different reference number or name, but you should see something similar to:

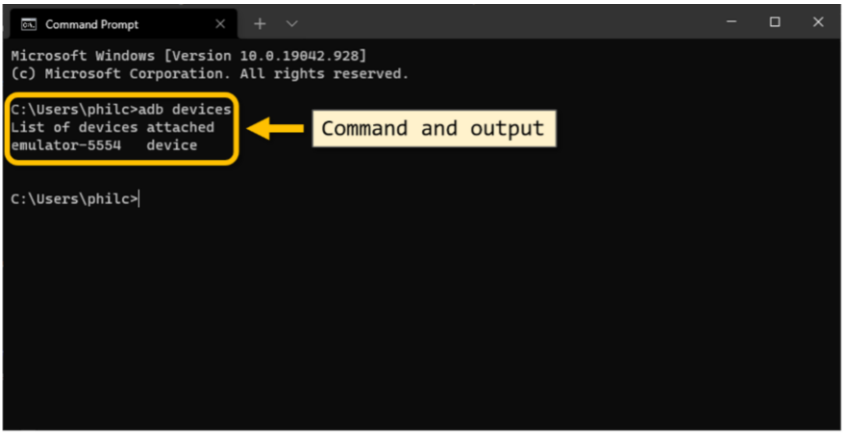


Fig. 8.17: ADB Walkthrough - “adb devices” Command

This has confirmed that your emulated device is visible to your computer via ADB, and we can proceed with pulling the data from the device.

Step 5: Get ADB Root

This is only really going to work as we are emulating our Android device and simply using the content for research purposes. However, usually you would have to use additional steps to pull the application data from a modern Android handset due to permission restrictions and device security settings. However, in the interests of speed and simplicity, type in the following command to your CMD window:

```
adb root
```

This will provide you with root access to the device via ADB, meaning we can bypass a lot of the existing security and protections.

Step 6: Find the Application Data

Here we shall navigate through the device to confirm the location of the application data. We would expect the find the app package and associated directories, located within the file path:

```
/data/data/<package ID>
```

In this example I have named the app "Tasky" and it has a package ID of:

```
com.tutorial.tasky
```

Yours may be different, depending on how you build the app and what name you gave it, so just bear this in mind when looking for the package. First, in your CMD window type the following commands in order and press enter/return at the end of each one:

```
adb shell
```

Then type:

```
ls
```

Next type the command:

```
cd /data/data
```

This takes us to the data folder where all of the packages are located. We could have done two separate steps of running the command `cd /data` twice in succession as there are two directories called "data", one nested within another. However, the command we used combined both into a single command. Next, use the `ls` command to locate your application package:

```
ls
```

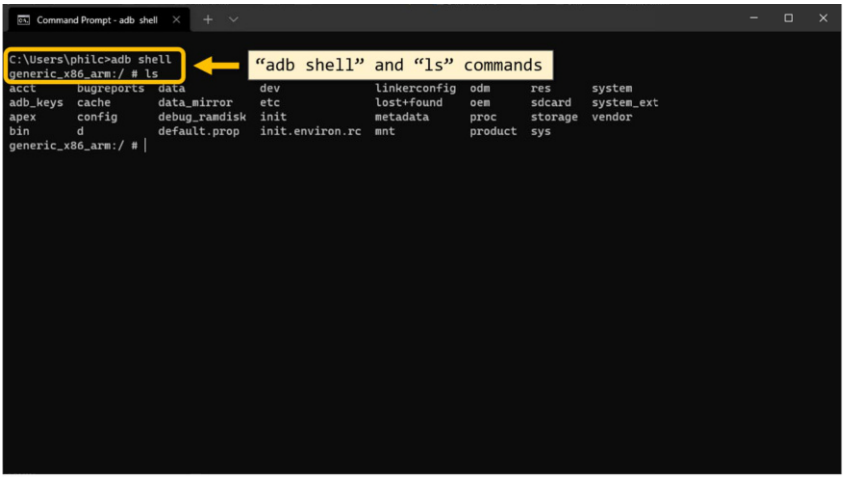


Fig. 8.18: ADB Walkthrough - "adb shell" and "ls" Commands

The list of packages will be displayed, and you can look through the list to find your application. In this example, the app we have created is located near the end of the list (Fig. 8.19):

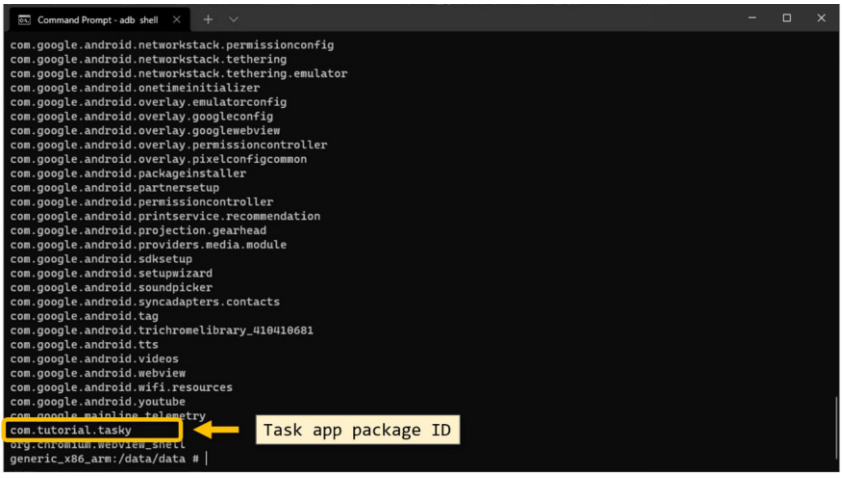


Fig. 8.19: ADB Walkthrough - Locate Package ID

We have now confirmed that our app package directory is located at:

`/data/data/com.tutorial.tasky`

Now, exit the adb shell using this command:

```
exit
```

Step 7: Use the "pull" Command

In this final step we use the "pull" command to pull a copy of the package directory out from the device and place the copy into a location of our choice on our computer. In your CMD window type the following command:

```
adb pull "/data/data/com.tutorial.tasky" "C:\Android"
```

Here we have specified the command `adb pull`, and provided what are known as `parameters`. The pull command can recognise several parameters, and we have provided both a target for our pull action, as well as a destination of where to place the copied content.

You should now be able to open the destination location on your computer and find the exported copies of the package folders. Within these folders you will find your initialised Realm database file.

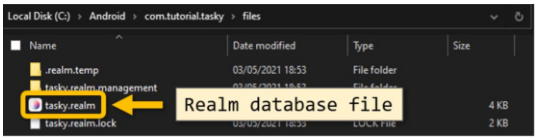


Fig. 8.20: ADB Walkthrough - Output Files

8.5.4 The Realm Database File Structure

As digital forensic examiners and investigators it is usually helpful to understand the inner workings of the artefacts that we analyse and decode. It is not always possible for us to fully reverse engineer these artefacts, but with research and testing we can often, as a community, begin to figure out some of the key and important hex strings and data blocks that reside within. With Realm this is no exception, and as the source code is publicly available, we have the benefit of being able to use this code to help identify how the database is structured at a byte level for some of those important components.

So far in this chapter we have discussed how Realm compares to traditional SQLite databases, how it differs, and have provided a brief overview of what to expect from a Realm database. We shall now take that understanding and dive deeper into the inner-workings of the files themselves, highlighting and identifying some key structures. We include this in the hopes that it can both direct examiners to relevant documentation to continue this research, as well as assist those who wish to begin creating scripts and other tools to begin parsing these databases themselves.

This section will guide you through the basic concepts of the Realm database file structure based on the implementation of the 'Realm Core' . The source code of the Realm Core implementation is available on Github at https://github.com/realm/realm-core [62]. At the time of writing, the source code referenced with this chapter relates to 'realm-java-v10.4.0'. The Realm Core is actively being developed which does mean that any static analysis of the source code may change over time.

Navigating to the Github repository directory `/src/realm/` we find many C++ source code files (*.cpp). We have analysed and researched some of these files to help identify some of the content for this section, identifying some structures and confirming some byte references and offsets. However, remember that this code is under active development, and therefore we advise examiners and investigators to validate and verify these findings, as is good practice, for all future versions of the source code.

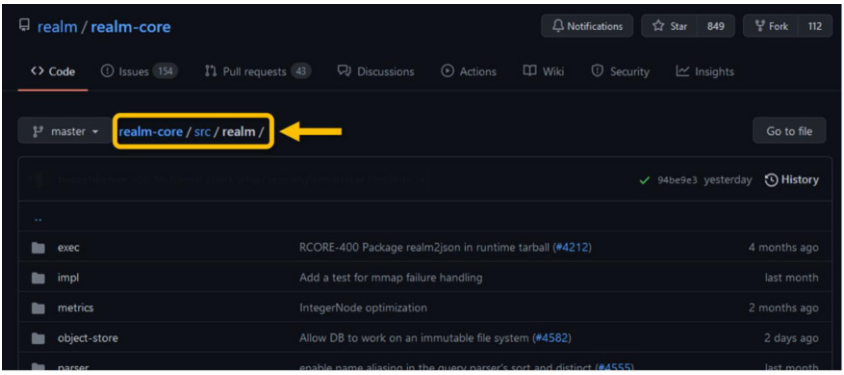


Fig. 8.21: Screenshot of the realm-core/src/realm/ directory

8.5.5 Realm File Header

Each Realm database contains a 24-byte header that can be broken down into component parts. We found the header to be defined within the file `alloc_slab.hpp` which is a form of header file, and can be found at https://github.com/realm/realm-core/blob/master/src/realm/alloc_slab.hpp [65]. The code relates to the definition of what is known as a struct, which is a C++ data structure where the term literally stands for "structure". It is used to store different elements of different data types within a fixed, structured environment, which is perfect for building a header of a set size and design. At the time of writing, the code was located at line 520 and reads

as follows:

```
// 24 bytes

    struct Header {

        uint64_t m_top_ref[2]; // 2 * 8 bytes

        // Info-block 8 bytes

        uint8_t m_mnemonic[4]; // "T-DB"

        uint8_t m_file_format[2]; // See 'library_file_format'

        uint8_t m_reserved;

        // bit 0 m_flags is used to select between the two top
            refs.

        uint8_t m_flags;
    };
```

Based on this declared Header struct, the 24-byte header contains a reference to a 'top ref', the 'mnemonic', a 'file format', 'reserved' and 'flags'. Each of these elements will be described below. However, in essence, the byte allocations are as follows:

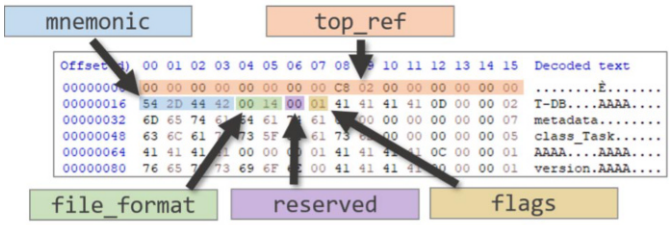


Fig. 8.22: Realm Header Structure

- **16 bytes**: 2 x 8 byte references to the 'top_ref'
- **4 bytes**: mnemonic / 'magic value'
- **2 bytes**: file_format
- **1 byte**: reserved
- **1 byte**: flags - bit 0 is used to select between the two top_ref pointers

"Top Ref" - Bytes 0x00 to 0x0F (d0–d15)

The top_ref element stands for "Top reference" and relates to the root of the database. The element is sixteen bytes in length, using the first sixteen bytes of the database

file, but is actually made up of two eight byte components (see Fig. 8.23). Both components are a top_ref but each eight byte string references an offset within the file, with the first referencing the start of the first top_ref, and the next eight byte string referencing the second top_ref.

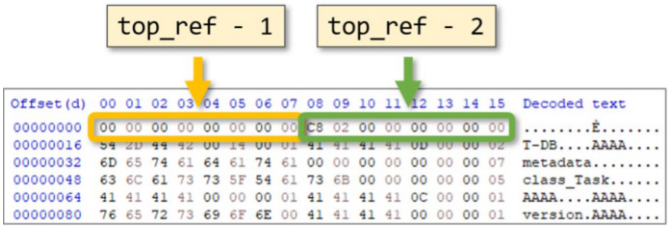


Fig. 8.23: Realm Header - top_ref

These references point to two separate arrays that act as the root nodes. These, in turn, point to two distinct branches that each link to a series of arrays, both branches seemingly mirrored (or almost mirrored). At the time of writing, our understanding based on testing suggests that the database utilises multiple branches through the top_ref mechanism as a form of journaling, alternating writes and commits between the two different root nodes. Starting at one of the two top_ref arrays, one can follow the references to other reference arrays to build up a tree. The current, most up to date state of the database is represented by rebuilding the Tree from the root node of the 'current top ref'. The database identifies the current top_ref through the flag byte, and writes new data to the other top_ref, preserving the current branch of arrays. This could be thought of as a form of WAL, where the "current" top_ref is like an SQLite database file and remains untouched, but the other referenced root node is used to write changes prior to any commits.

However, similar to an SQLite database where WAL checkpoint rules do not always appear to be followed by the database itself, the rules governing changes with Realm arrays also appear to be fairly flexible and not always consistent.

"Mnemonic" - Bytes 0x10 to 0x13 (d16–d19)

These four bytes contain the ASCII value 'T-DB' which is called the 'mnemonic' of the Realm file. In other words, bytes 0x10 to 0x14 contain the magic value of the Realm file.

At the time of writing the mnemonic values are static and always equate to the ASCII "T-DB". This may change with future iterations of the database code, but currently this is a very good way of being able to immediately identify the file as a Realm database, and enables examiners to utilise the hex string 0x542D4442 in searches when seeking to find Realm database amongst datasets.

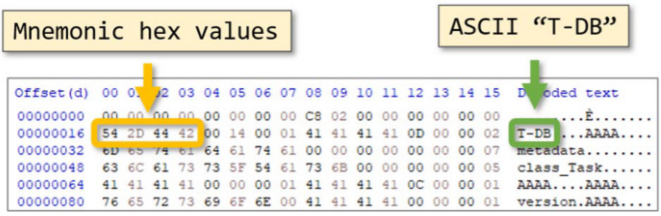


Fig. 8.24: Realm Header - mnemonic

“File Format” - Bytes 0x14 to 0x15 (d20–d21)

These two bytes are described in the Realm core source code as the ‘file format’. Both bytes form an integer value and represent the version number.

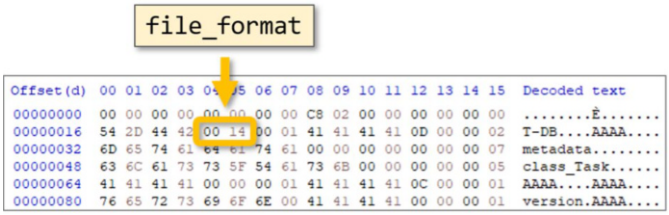


Fig. 8.25: Realm Header - file_format

In the documentation the file format is actually also referred to as the “version”, and in the `alloc_slab.cpp` file [63] there is actually reference to the variable `file_format_version` variable, along with an object method of `get_committed_file_format_version()` `get_committed_file_format_version()`. In this method, it was observed that the file format may be updated to 0x14 (d20) whenever the process (called ‘session’ in the Realm Core source code) accessing the Realm database requires so. This functionality is likely built in to provide compatibility with processes that handle newer/future file formats. For example, in the `group.cpp` file, the object method `read_only_version_check()` requires a file format upgrade if the `file_format_version` is lower than 0x14 (d20) when the database needs to be opened in read only mode. At the time of writing the maximum value for the file format was 0x14 (d20) but may be subject to change in the future.

The file `group.cpp` of the source code contains an object method called void-Group::open(). In this method, the `target_file_format_version` is assigned to a value determined by `get_target_file_format_version_for_session()`, suggesting the desired file format version of the file itself is determined by the current process which calls `open()`. The `open()` object method only returns without errors if the `target_file_format_version` is 0 or equal to the file format version.

The file format version is assigned to 0 upon the creation of an empty database file where the Realm file header needs to be initialised. The initialisation of an empty header is defined in `alloc_slab.cpp` as follows:

```
const SlabAlloc::Header SlabAlloc::empty_file_header = {

    {0, 0}, // top-refs
    {'T', '-', 'D', 'B'},
    {0, 0}, // undecided file format
    0,      // reserved
    0       // flags (lsb is select bit)
};
```

"Reserved" - Byte 0x16 (d22)

The reserved byte, at the time of writing, is always set to zero, and has never changed throughout testing. This byte currently appears to be unused, as the name suggests, but may be utilised.

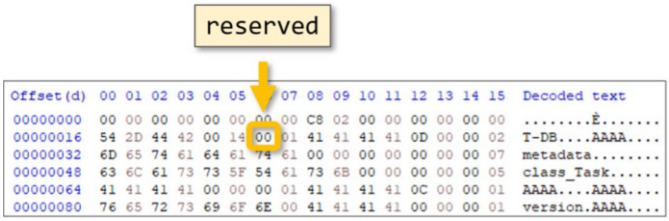


Fig. 8.26: Realm Header - reserved

"Flags" - Byte 0x17 (d23)

The final byte value of the header represents flags. The first bit (the least significant bit) of the last byte indicates which top reference is currently active. If this bit is set to 0, the first top reference is currently active and when this bit is set to 1, the second top reference is active. The other seven bits of the last byte are at the time of writing unused.

In Fig. 8.27 we can see that the byte value is 0x01, meaning that the last bit must be a 1 (the binary breakdown of 0x01 being b0000001), indicating that in this example the current referenced array branch is top_ref 2 (Fig. 8.28).

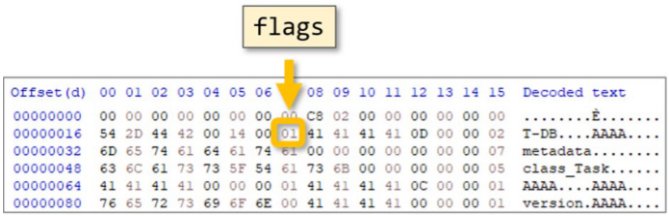


Fig. 8.27: Realm Header - flags

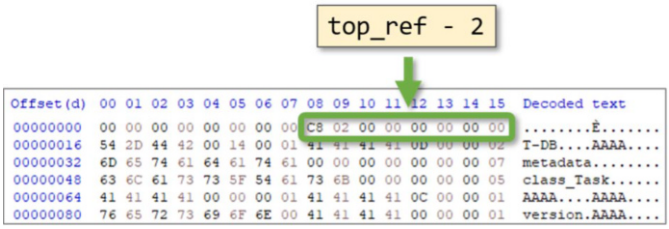


Fig. 8.28: Realm Header - top_ref 2 example from flag value of 0x01

8.5.6 Realm File Arrays

After the Realm header a *.realm file is made-up entirely of arrays in a B+tree format, with a single root node, inner nodes that act as sign posts, and leaf nodes that generally contain the data of interest to investigators.

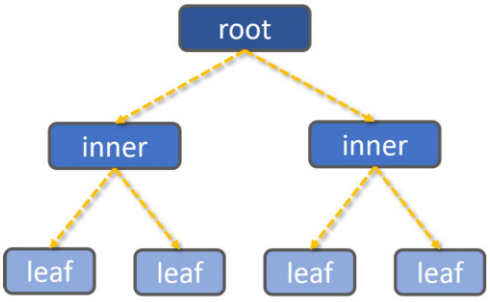
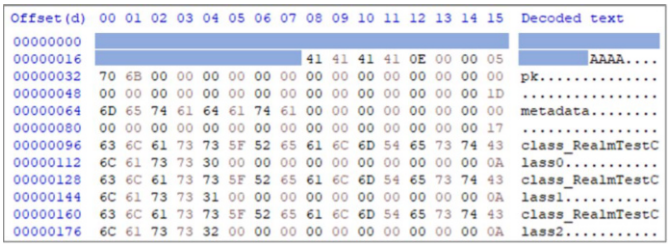


Fig. 8.29: Example Node Structure

The top_ref header information points to offsets within the file where you will find (typically) “reference arrays”. These reference arrays point to other nodes and arrays within the database, some of which will be other reference arrays, and some will be what are known as “data arrays”. These data arrays are where the actual core object data is stored.

Opening up a copy of the downloadable “demo.realm” file, available from realm.io, we will use the first array found after the Realm header, as an example of how arrays are structured and can be broken down (see Fig. 8.30).



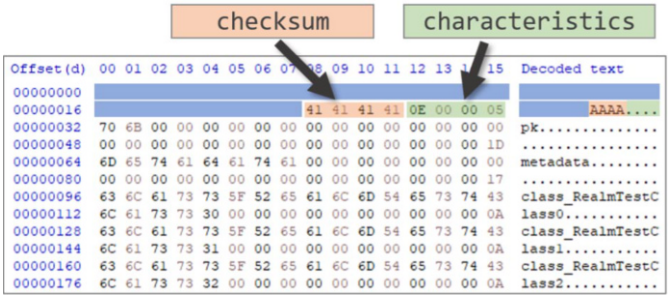
Offset (d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Decoded text
00000000																	
00000016									41	41	41	41	0E	00	00	05	AAAA....
00000032	70	6B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	pk..............
00000048	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	1D	....
00000064	6D	65	74	61	64	61	74	61	00	00	00	00	00	00	00	00	metadata........
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	17	....
00000096	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000112	6C	61	73	73	30	00	00	00	00	00	00	00	00	00	00	0A	lass0...........
00000128	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000144	6C	61	73	73	31	00	00	00	00	00	00	00	00	00	00	0A	lass1...........
00000160	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000176	6C	61	73	73	32	00	00	00	00	00	00	00	00	00	00	0A	lass2...........

Fig. 8.30: Example Node Structure

8.5.7 Realm Array Header

Every array within Realm starts with an 8-byte header broken into two distinct parts:

1. Checksum value (4 bytes)
2. Array characteristics (4 bytes)



Offset (d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Decoded text
00000000																	
00000016									41	41	41	41	0E	00	00	05	AAAA....
00000032	70	6B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	pk..............
00000048	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	1D	....
00000064	6D	65	74	61	64	61	74	61	00	00	00	00	00	00	00	00	metadata........
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	17	....
00000096	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000112	6C	61	73	73	30	00	00	00	00	00	00	00	00	00	00	0A	lass0...........
00000128	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000144	6C	61	73	73	31	00	00	00	00	00	00	00	00	00	00	0A	lass1...........
00000160	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000176	6C	61	73	73	32	00	00	00	00	00	00	00	00	00	00	0A	lass2...........

Fig. 8.31: Realm Array Header Example

This can actually be broken down further into the following components:

Table 8.3: Table of Realm Array Header Components

Header Section	Offset	Size	Description
Checksum	0x00 (0)	4 bytes	Checksum (dummy) (“AAAA” in ASCII)
Characteristics	0x04 (4)	1 byte	Flags
Characteristics	0x05 (5)	3 bytes	Size

8.5.8 Checksum

The first four bytes contain a checksum value that, at the time of writing, is only considered a “Dummy Checksum” within documentation, suggesting that this component may change in future iterations of the code. The checksum consists of four matching byte values, namely 0x41414141, which reads as “AAAA” in ASCII. The source code shows us that these four ASCII characters are used when scanning for arrays within the database [69].

8.5.9 Flags

The fifth byte of the array header represents flags and utilises several bit groupings to denote different configurations for the array [66]. This means that to read or parse the bit groupings we need to breakdown the byte into 8-bits and identify which bits represent which groups. The breakdown is as follows:

Table 8.4: Breakdown of “Flags” byte into bit groupings

Bit	Group	Description
1	1	is_inner_bptree_node
2	2	has_refs
3	3	context_flag
4 5	4	width_scheme
6 7 8	5	width_ndx

Bit Group 1: is_inner_bptree_node

The first bit of the flags byte indicates whether a Realm array is an 'inner node'. A value of 1 would indicate that the node is an inner node, which would mean that the array must be a reference array, as opposed to a data array. While analysing several Realm database files, none of the Realm arrays had the `is_inner_bptree_node` flag set. One reason for not using the `is_inner_bptree_node` flag in the Realm array header could be that the tree structure of the Realm database can still be constructed without checking this flag: starting from the root node (the top_ref array), and following all references of the tree until all identified references have been exhausted and pursued, allows a researcher to build the tree manually. Take a look at the code snippet below, which comes from the file `array.cpp`:

```
void Array::set_type(Type type)
{
    REALM_ASSERT(is_attached());

    copy_on_write(); // Throws

    bool init_is_inner_bptree_node = false, init_has_refs =
        false;
    switch (type) {
        case type_Normal:
            break;
        case type_InnerBptreeNode:
            init_is_inner_bptree_node = true;
            init_has_refs = true;
            break;
        case type_HasRefs:
            init_has_refs = true;
            break;
    }
    m_is_inner_bptree_node = init_is_inner_bptree_node;
    m_has_refs = init_has_refs;

    char* header = get_header();
    set_is_inner_bptree_node_in_header(init_is_inner_bptree_node,
        header);
    set_hasrefs_in_header(init_has_refs, header);
}
```

It appears an array with the `is_inner_bptree_node` flag set, also sets the bit flag for `has_refs`. Therefore, it is concluded that any array that is considered an `inner_bptree_node`, also contains references to other arrays.

Bit Group 2: has_refs

A Realm Array header where the second bit of the flags byte is set to 1, indicates the Realm Array contains references to other Realm Arrays in its payload, making it a "reference array". This, in turn, makes the array the parent of any other arrays that

it directly references. The payload of any reference array will typically consist of elements that store pointers to the child arrays. These pointers will be integer values that directly correspond to an offset in the file. These offsets always consist of 8-byte strings.

Bit Group 3: context_flag

The third bit within the flags byte is known as the "context_flag". However, rarely set to 1, and the full purpose of the flag remains unclear. Code from the `array.hpp` file [66] enables us to deduce that the context flag can be used to tell what type of leaf node the given array is. Unfortunately, there is not much more information currently shared regarding the 'context flag' in the Realm Core source code at the time of writing.

Bit Group 4: width_scheme

The array header contains information that enables us to calculate the total size, in bytes, of the payload held within the array. This calculation is done by using both the `width_scheme` and `width_ndx` bit groupings. We can therefore calculate the total size of the array by identifying the values held within these two bit groups.

The `width_scheme` consists of two bits which are added together to create a integer value. This type of calculation, therefore, allows for three possible value outcomes: 0, 1, or 2. Depending on the value, the payload is calculated in a certain way, as defined in `node_header.hpp` [67]. The following code snippet outlines the calculations and intentions:

```
static void set_wtype_in_header(WidthType value, char* header)
    noexcept
    {
        // Indicates how to calculate size in bytes based on
            width
        // 0: bits       (width/8) * size
        // 1: multiply   width * size
        // 2: ignore     1 * size

        typedef unsigned char uchar;
        uchar* h = reintercept_cast<uchar*>(header);
        h[4] = uchar((int(h[4]) & ~0x18) | int(value) << 3);
    }
```

Therefore, the `width_scheme` could be translated as per Table 8.5. The 'size' in the calculation is the value represented in the last three bytes of the Realm Array header. The value of 'width' in the calculation is represented in bit group 5, after applying the width translation table (see Table 8.6 below).

Table 8.5: Calculations Required for width_scheme Values

Value of width_scheme	Meaning	Calculation for array payload
0	Calculate size with number of bits	Cell(width*size/8)
1	Calculate size with number of bytes	Width*size
2	Ignore width in size calculation	size

Bit Group 5: width_ndx

Bits 6, 7, and 8 of the `flags` byte form 'bit group 5', referred to as `width_ndx`. These three bits collectively are used to represent the values 0 to 7 (7 being when all three bits are set to 1), and are used to indicate the value of 'width'. With the translation table below, the translation from `width_ndx` to the actual value of 'width' can be found. The value "width" represents the number of elements that are contained within the Realm Array payload.

Table 8.6: Translation table for width_ndx

width_ndx calculated value	0	1	2	3	4	5	6	7
Value of 'width'	0	1	2	4	8	16	32	64

8.5.10 Size

The size of the Realm Array payload is described in the last 3 bytes of the Realm Array Header. The size represents the amount of bits or bytes of one element in a Realm array. The value of `width_scheme` determines how the size value is used to calculate the overall payload. Since the amount of elements is represented in `width_ndx`, the size of each element can be calculated by using the knowledge found in the `width_scheme` along with the value of size, one can calculate the complete size of a payload, and thus calculate the total size of the array. So to summarise:

Table 8.7: Calculations Required for width_scheme Values

Width_scheme	The method used to calculate the overall size of the payload
Width_ndx	The number of elements present within the payload
Size	The size of each element within the payload

8.5.11 Realm Array Payload

After the 8-byte header, the remainder of the Realm Array is followed by multiples of 8-bytes, which makes up the "payload". The amount of bytes that follow after the Realm Array Header (so, the total size of the payload) can be calculated by taking the `width_ndx`, `width_scheme` and size as explained in the section above.

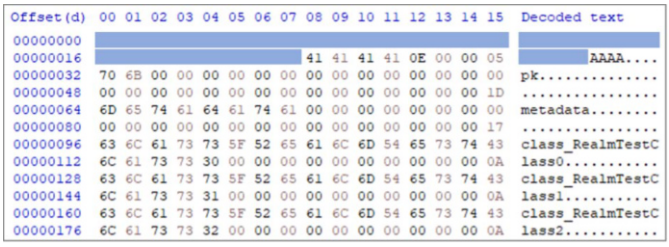
This is outlined within the `node_header.hpp` file found within the Realm-core documentation [67] where you will find the following code:

```
static size_t calc_byte_size(WidthType wtype, size_t size,
    uint_least8_t width) noexcept
    {
        size_t num_bytes = 0;
        switch (wtype) {
            case wtype_Bits: {
                // Current assumption is that size is at most
                    2^24 and that width is at most 64.
                // In that case the following will never
                    overflow. (Assuming that size_t is at least
                    32 bits)
                REALM_ASSERT_3(size, <, 0x1000000);
                size_t num_bits = size * width;
                num_bytes = (num_bits + 7) >> 3;
                break;
            }
            case wtype_Multiply: {
                num_bytes = size * width;
                break;
            }
            case wtype_Ignore:
                num_bytes = size;
                break;
        }

        // Ensure 8-byte alignment
        num_bytes = (num_bytes + 7) & ~size_t(7);

        num_bytes += header_size;

        return num_bytes;
    }
```

From this code-snippet we can deduce that if we take the value of size and multiply it with the width value, these are the amount of bits or bytes of the complete Realm Array payload. Whether this value is in bits or bytes depends on the `width_scheme` (referred to by wtype in the code example). This number is padded until it becomes a multiple of 8. The total size of a Realm Array is the payload size in addition to the Realm Array Header size (which is 8 bytes).

8.5.12 Size Calculation Example

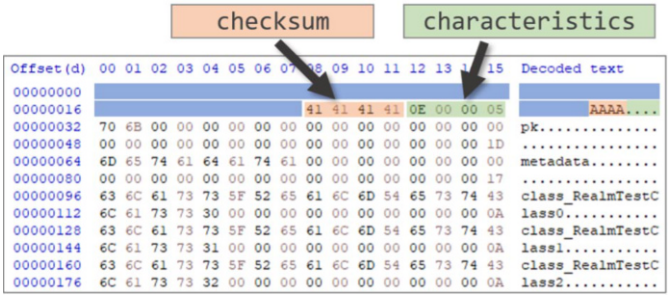
We shall revisit the example content used when introducing the Realm Array Header in 8.5.7 above, to demonstrate how some of these calculations work.



Offset (d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Decoded text
00000000																	
00000016									41	41	41	41	0E	00	00	05	AAAA....
00000032	70	6B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	pk..............
00000048	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	1D	....
00000064	6D	65	74	61	64	61	74	61	00	00	00	00	00	00	00	00	metadata........
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	17	....
00000096	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000112	6C	61	73	73	30	00	00	00	00	00	00	00	00	00	00	0A	lass0...........
00000128	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000144	6C	61	73	73	31	00	00	00	00	00	00	00	00	00	00	0A	lass1...........
00000160	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000176	6C	61	73	73	32	00	00	00	00	00	00	00	00	00	00	0A	lass2...........

Fig. 8.32: Realm Array Example

This example looks at an Array that is stored immediate after the Realm file header, which we have blanked out to help focus on the Array itself (Fig. 8.32). Remember, our Array Header consists of eight bytes that can be broken down into two distinct elements (Fig. 8.33).



Offset (d)	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Decoded text
00000000																	
00000016									41	41	41	41	0E	00	00	05	AAAA....
00000032	70	6B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	pk..............
00000048	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	1D	....
00000064	6D	65	74	61	64	61	74	61	00	00	00	00	00	00	00	00	metadata........
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	17	....
00000096	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000112	6C	61	73	73	30	00	00	00	00	00	00	00	00	00	00	0A	lass0...........
00000128	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000144	6C	61	73	73	31	00	00	00	00	00	00	00	00	00	00	0A	lass1...........
00000160	63	6C	61	73	73	5F	52	65	61	6C	6D	54	65	73	74	43	class_RealmTestC
00000176	6C	61	73	73	32	00	00	00	00	00	00	00	00	00	00	0A	lass2...........

Fig. 8.33: Realm Array Header Example

We can see that the header for our array is made up of the following eight bytes:

Table 8.8: Array Header Bytes

Byte	1	2	3	4	5	6	7	8
Hex Value	41	41	41	41	0E	00	00	05

8.5.13 Array Example - Header

The first four bytes are our checksum, which, at the time of writing, is always 0x41414141 ("AAAA" in ASCII) and then our next set of four bytes are the characteristics values. If you recall from earlier in the chapter, this is further broken down into two different components, namely the flags and size values.

8.5.14 Array Example - Flags

The fifth byte is our flags, which is currently the value 0x0E, which is the binary value of b00001110. If you recall, the flags byte is broken into five "Bit Groups" that represent different things depending on their values. For our example value, we can breakdown the byte as follows:

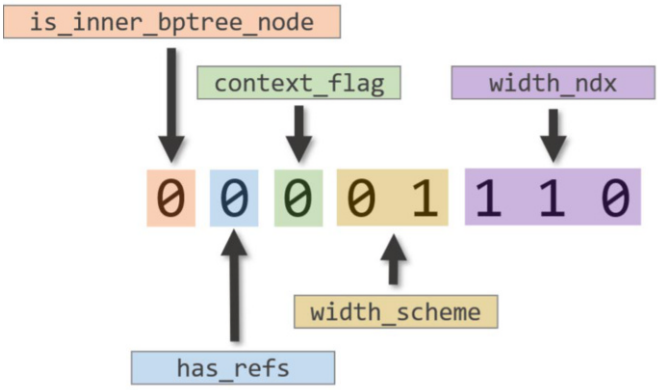


Fig. 8.34: Realm Array Example - Header Breakdown

From this we can identify the following:

- Our array is not an "Inner Node", given our `is_inner_bptree_node` value is set at 0.
- Our array is likely a "Data Array" as the `has_refs` value is set at 0.
- Our value of `width_scheme` is set at 1. Looking back at Table 8.5 in 8.5.9 above, this tells us that the scheme we need to use is calculating with the number of bytes (Width*size).
- The value of `width_ndx` is 110 in binary, which is the value 6 in decimal, which gives us the `width_ndx` value of 32 from the translation table provided by Table 8.6 in section 8.5.9 above.

To clarify how we made this conversion, we followed these simple steps:

1. Convert the Binary Value 110 into Decimal

This can be done using a calculator or converter, but to manually do this conversion we can simply look at the base 2 number values:

Table 8.9: Calculations Required for width_scheme Values

Base 2 columns	4	2	1
Binary Value	1	1	0

So our decimal calculation is:

$$1 \times 0 = 0$$
$$2 \times 1 = 2$$
$$4 \times 1 = 4$$
$$0 + 2 + 4 = 6$$

2. Find the Decimal Value on the Translation Table

Fig. 8.35: Annotated Copy of the width_ndx Translation Table

3. Identify the Given Value that the Table Returns

As we can see above, the returned value for "width" is 32.

8.5.15 Array Example - Size

Our final series of bytes within the Array Header denotes the value for size. Given we have a three-byte value of 0x000005, which is the decimal value of 5, we can confirm the value of size to be 5.

Now that we have our values for both "width" and "size" we can use the given `width_scheme` calculation method to identify the total size of the payload for this array, as follows:

Size x Width = Payload size in bytes
5 x 32 = 160 bytes

Remember that the total size of our array is the header plus the payload size. Our header is always going to be 8 bytes long, and our payload is 160 bytes, so this array should be 168 bytes in total. You can see in the following screenshot of our Realm file viewed through the software tool HxD [36] that our array is, indeed, 168 bytes in length (Fig. 8.36):

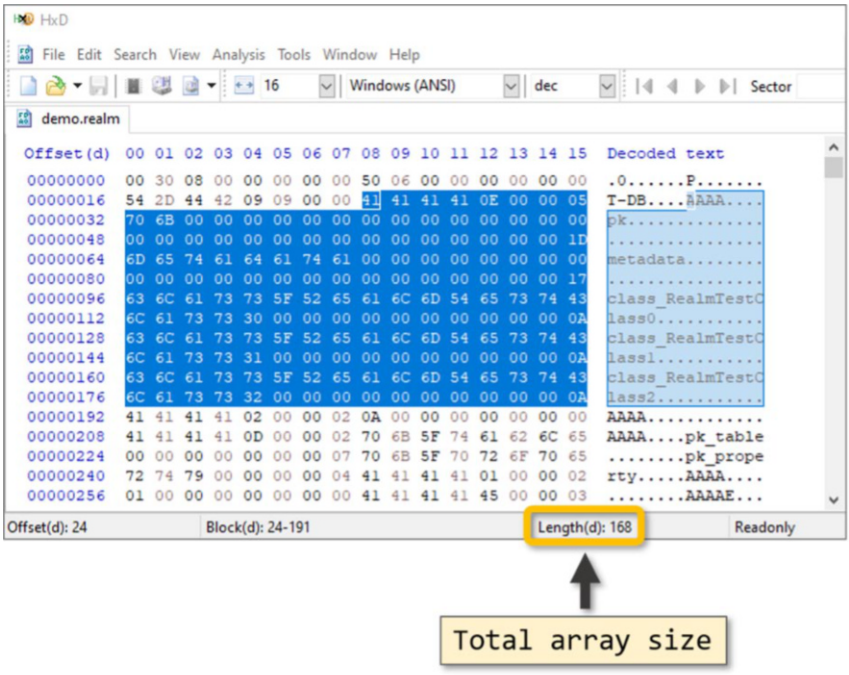


Fig. 8.36: Screenshot of Realm File in HxD showing Total Size of an Array

If you look closely at the end of the array in Fig. 8.36, you will notice that the byte following the end of the highlighted array is the start of another array, with the array header ASCII values “AAAA” clearly visible within the decoded text.

8.6 Conclusion

Realm databases have been considered, by some, to be the new format that will ultimately replace SQLite. While we have seen some evidence of this with some applications, the anticipated change has been far slower than originally believed, and

Realm databases are still not commonly found within the most common applications examined by Digital Forensic examiners. This may change in the future as momentum builds for the database, or the movement may continue to struggle to gain traction as developers find workarounds and clever ways to continue implementing the more widely known and understood SQLite format.

The Realm database format has, at the time of writing, an active developer community with an ever-evolving code-base. The format is still relatively new, with a number of areas of code still using "dummy" or "temporary" values or data structures while the code is developed. The open-source nature of the realm-core code enables forensic examiners to reverse engineer the code in order to discover exactly how these database are structured and operate, which may be vital in digital forensic investigations.

However, the move away from SQL structures and into editable and customisable object-oriented code also adds challenges for forensic examiners, especially when every database could, in theory, be coded to operate and function is very different ways from any other. Understanding the fundamental concepts and structures behind Realm databases should enable examiners to navigate and understand some of the core data structures, even if the object functions remain challenging to decode and decipher.

This chapter has aimed to introduce some of the core fundamentals behind the Realm database, with a view of providing the foundations and tools that could be helpful in continued, further research and analysis. We hope that readers are able to use any knowledge gained from this chapter and referenced materials, to continue to explore and build their understanding of this database format, and we hope that the community continues to explore, share knowledge, and help one another enhance our understanding of new and developing file formats and structures for the benefit of all.

Chapter 9
Protocol Buffers

Chris Currier

Abstract Protocol Buffers (Protobufs) are discussed in this chapter, from creating one to analyzing the data. This particular serialization format, originally developed by Google, is used in various apps. We discuss creating a protocol buffer and adding data through Python step by step. This provides a better understanding of how and why protocol buffers are formed and used. We also clarify how to recognize and decode them during a forensic examination.

9.1 Introduction

I remember being on the edge of my seat as Johan Persson, a developer at MSAB, first introduced me to Protocol Buffers. Why? *Protobufs*, as they are commonly referred to, contain data that we as examiners may find helpful in an investigation. I had no idea how to find them and view the payload they carried. However, that was to change quickly.

9.1.1 What is a Protocol Buffer?

A Protocol Buffer provides a format for taking compiled data (many different languages/platforms supported) and serializing it by turning it into bytes represented in decimal values. This makes the data smaller and faster to send over the wire. We call this *serialization* in computer science.

A protocol buffer is a data format structured in a very efficient binary format. The structure is defined in a *.proto* file, which is in a readable text format. The concept is similar to XML, where the schema description can be done inline or in a separate

Chris Currier
MSAB, Hornsbruksgatan 28 SE-117 34 Stockholm Sweden e-mail: chris.currier@msab.com

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_9

file. The *.proto* file is then used to generate code for reading from and writing data to the protocol buffer. Due to its nature, protocol buffer data is very suitable for transmission over networks. When transmitted over networks, it is often compressed with GZIP to minimize the size of the data. The protocol buffer concept was created and is used extensively by Google. Other users of protocol buffers include Apple and app developers.

So, where do we begin? Of course, the best place to learn about Google's Protocol Buffers is Google (see Fig. 9.1). Google defines how to structure and use Protocol Buffers [24]. The structured or rigid format used by Protocol Buffers is often referred to as a *schema*.

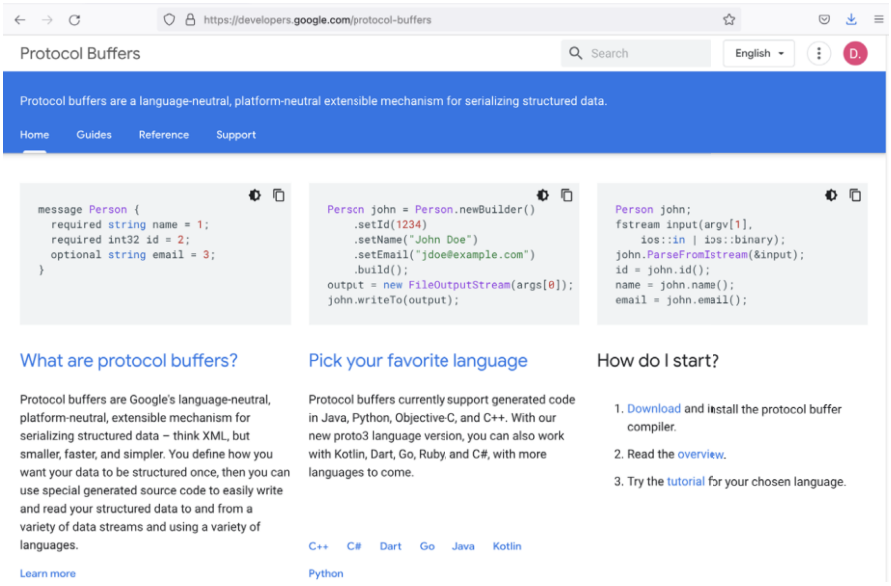


Fig. 9.1: Google's Protocol Buffers

Google itself says about its format in the developer documentation that it is a platform-independent, language-independent and easily extensible serialization format. Of course, other formats allow serialization of data too. Java-Serial is an example of this. Unlike this, Protobuf is a language-independent transmission format. Also, XML would be an option. However, Protobuf is smaller and faster than most of the other formats [24]. A significant advantage of Protobuf is that we only need to define the structure for the data to be transferred once and can then exchange it over a wide variety of data channels. The programming language is secondary since we are language-neutral. The data stream itself (network or file) is also irrelevant. The definition of the protobuf message always remains the same [24].

9.1.2 Why are Protocol Buffers Used?

Now think of a network with data being transmitted through it. How do we get data through the network faster? The smaller the data, the faster it will be. We also do not need it to be human-readable during transmission. This is where Protocol Buffers shine with faster transmission. Figures 9.2 and 9.3 demonstrate the time and size of Protbufs, based on tests performed to consider encoding and decoding benchmarks and common browsers [58, 43].

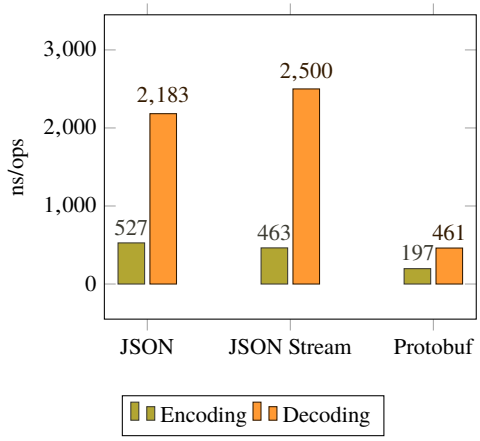


Fig. 9.2: Encoding and Decoding Performance of Protobufs [58]

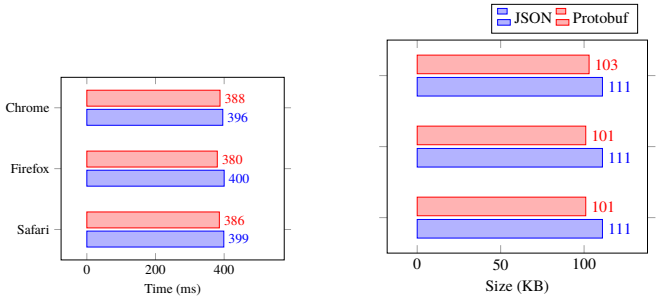


Fig. 9.3: Compression Environment of Protobufs and JSON [43]

The authors of a benchmark study in [43] conclude that ProtoBuf performs significantly better than JSON. The messages are significantly smaller and are transmitted

much faster at the same time. However, there is always someone faster, and that brings us to the term FlatBuffers:

"Protocol Buffers is indeed relatively similar to FlatBuffers, with the primary difference being that FlatBuffers does not need a parsing/ unpacking step to a secondary representation before you can access data, often coupled with per-object memory allocation. The code is an order of magnitude bigger, too. Protocol Buffers has neither optional text import/export nor schema language features." [15]

As you can see, it is not just about speed but also the size of the data. The protocol buffer is not only the code but also the key. The data sent is binary and can be converted and looked at. Different languages (code) may be supported and used to enter and view this data.

9.2 Using Protocol Buffers

This section will clarify how ProtoBuf works and what data is needed. For this, the first step is to generate a description of the message types used and the access service:

Messages

To create a ProtoBuf message, we must first create an appropriate template. This template is usually saved in a file with the extension *.proto*. The file is set up and used alongside another programming language such as Python. The data (or user data) can be added using the same scheme (Field assignments: Type, Name, Tag) and then sent internally or externally over the wire. Google set up Protocol Buffers for their internal communication. Data is transmitted as binary. For this reason, we can encounter it in almost every Google app.

Services

Protocol Buffers are not just about messages but also services. For this reason, we need a service description. It describes the interface of the methods offered via the service. If we want to create an RPC service using a proto buffer, then the service must be given a name under which it can later be called once. In this case, the developer documentation recommends formulating both the service name and the access methods in CamelCase (with an initial capital). Here we have a brief example of defining a chat service that provides precisely one access method:

```
service ChatService{
    rpc GetChats(ChatRequest) returns (CharResponse);
}
```

One such service is Google's *gRPC*. These RPC (Remote Procedure Calls) methods accept a request "message" and return a response "message". Protobuf is often used with HTTP and RPCs for local and remote client-server communication. Protobuf is used for the description of the required interfaces and message types. The protocol composition is also summarized under the name gRPC [30]. In this case, the call to the remote method - encapsulated by a service - is provided platform-independently via a service description. The steps for generating a protobuf message and subsequent transmission are briefly summarised again in Fig. 9.4. The message data (message refers to type or object and not to chat or text) is then transmitted over the wire (internal or external). The supported platforms can all have code generated to deserialize the data and make sense of it, regardless of what coding platform created the data.

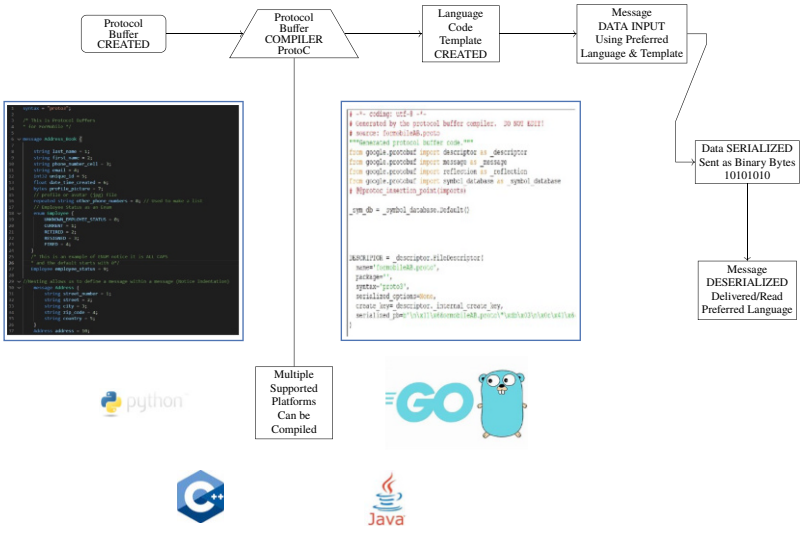


Fig. 9.4: How Protobufs Work

Looking at the data, we may not make sense of it without having the original code to know what the data represents. Unfortunately, we will not find the schema to help us make sense of the fields. So, let us look at the code to start with. That will give us a better understanding of what we see during the examination. Therefore, we will first discuss the design and implementation of a proto buffer using two small examples before turning to the forensic analysis of these special artefacts.

The Proto File

We start by creating a *.proto* file. We like to think of this file as the key or legend to the data we will see on a mobile device. Unfortunately, we will not find this key

or legend on the device itself as examiners. The *.proto* file is not included with the binary, with a few exceptions. As we will see, this often leaves some room for interpretation when we deserialize a Protocol Buffer manually. We will walk you through an example of the process. This example (see Fig. 9.5) will be for an Address Book, so we will name it *formobileAB.proto*. The file can be created with any editor.

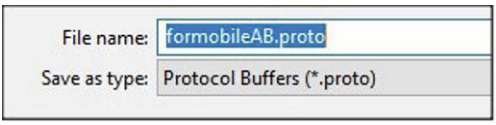


Fig. 9.5: FORMOBILE Protobuf Example

Define the Syntax

Google made Protocol Buffers public in 2008. In 2016 Google published Protocol Buffers 3. Since there is a different version, we have to identify which version of Protocol Buffers will be used. So, the first line of code needs to state this. Version 3 is used in these examples, as shown below.

```
syntax = "proto3";
```

Message Type

Now the message needs to be defined or named. This "message" is a code term that refers to the data and is not confused with terms chat or SMS text messages. The name should reflect the type of message based on content. The term *address book* should define our Protocol Buffer just fine. When using two words they use CamelCase and form one word i.e. *addressBook* or *AddressBook*, as shown below.

```
syntax = "proto3";

message AddressBook {

}
```

Fields

To create the first property, we need to know the type of data [int32], followed by property name [thread] and then identify it as the sequential property [1]. Fields

identify these data characteristics through Field Type, Field Name, and a Field Tag (or also called a *Field Number*). This is where we define the class characteristics that will be used. We consider what information would we want to know about, for example, a:

- Person
- Contact
- Web Broswer Search
- Location and a Chat

! Attention

Remember, the idea behind Protocol Buffers is to take code from another language and package it into a smaller container. This starts defining the data by naming it and then following with fields.

The actual data such as: *John Smith 40 1.85 90.71 Brown Blue* will not exist in this `proto` file, but in another file. This should remind you of how meta data type looks like in a chat message. Fig. 9.6 is an example found in an address book showing the fields and associated data and profile picture.

Scalar Values

A message is normally composed of a number of different scalar values. Each value is assigned to a particular type. Looking at an SQLite database table definition, we will find terms such as Integer, Boolean, Float, and String. These define data types. Protocol Buffers use these as well (see table 9.1). Since we usually want to exchange messages between different applications, the data they contain must be preserved. As we can see in the table below, Protocol Buffers are easily used with other programming languages: C++, Java, Python, and Go. Accordingly, we can easily map the data type of a programming language to a ProtoBuf type and vice versa. For more information about types and unsigned bit integers please refer to [26].

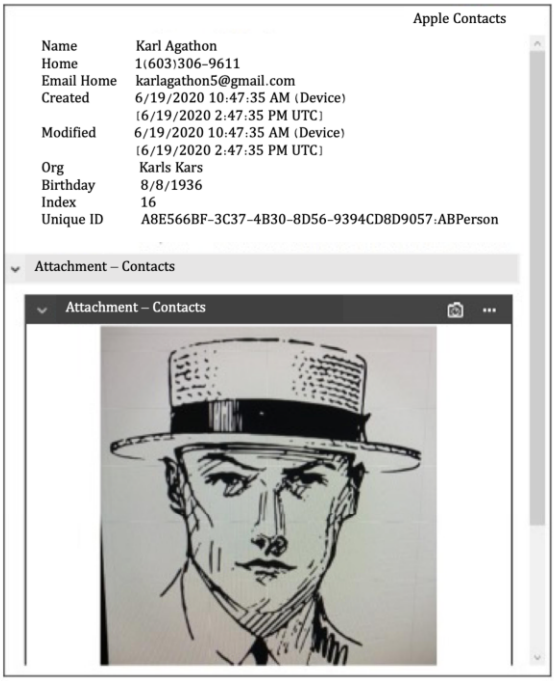


Fig. 9.6: Address Book Profile Example

`.proto` Type	Notes	C++	Java Type	Python Type	Go Type
double		double	double	float	*float64
float		float	float	float	*float32
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	*int32
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long	*int64
uint32	Uses variable-length encoding.	uint32	int	int/long	*unint32
uint64	Uses variable-length encoding.	uint64	long	int/long	*uint64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	*int32
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long	*int64
sfixed32	Always four bytes.	int32	int	int	*int32
sfixed64	Always eight bytes.	int64	long	int/long	*int64
bool		bool	boolean	bool	*bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	unicode (Py2) or str (Py3)	*string
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	bytes	[]byte

Table 9.1: Mapping Table for possible Scalar Types (Detail) [26]

! Attention

In this chapter, we will discuss several examples of Proto Buffers: (1) an address book, (2) a chat message, and (3) an Apple Maps example. You will find most of the files mentioned here: `www.github.com/Xamnr/ProtocolBuffers`. If you like, you can analyze the examples discussed here yourself. Just give it a try.

9.2.1 The Schema Defintion

The Protocol Buffer defines the Object (type of data and position). Not the actual data or user data. The actual data will be coded in Python or another language format. However, the type of data that will go into these fields needs to be defined. Protocol Buffers use fields. There are three types of fields:

- Field Type
- Field Name
- Field Tag (or Number)

Field Type

The field type uses the scalar values to define the type of data like *integer*, *string*, or *bool*. Thinking back to our Apple contact, shown below (Fig. 9.7), we have the following fields and data to consider:

- Name or maybe Last Name and First Name
- Phone Number (Home, Work, Cell)
- Email (Home, Work)
- A Unique Identifier
- A Profile Picture

If not set, specified, or unknown, every field will have a default value. Protocol Buffers do not recognize required fields. Instead, the runtime environment of the programming language is responsible for that, i.e. Java, Python, Go. This means that a field whose assignment corresponds to the default value is not serialized. It is just left out. Since the field is missing in the data stream, the receiver side automatically uses the default value in this case. This property, which may seem confusing at first glance, ensures that no unnecessary values are transmitted. The serialized data on the wire remains small.

Fig. 9.7: Field Type and Scalar Values (example)

VALUE	DEFAULT
Bool	False
Number	0
String (UTF-8 or 7-bit ASCII)	Empty String
Byte (or byte array)	Empty Bytes (or empty byte array
Enum	First Defined Value 0
Repeated	Empty List

Table 9.2: Field Type Default Values

Field Names

The field name represents one particular element within the message, therefore identifying the data for us. To identify a contact, we may use identifiers such as *last_name, first_name, phone_number_cell, email, unique_id, date_time_created, profile_picture*. When multiple words are used, each word is separated by an underscore "_". Again, we do not add the data such as *Karl Agathon*. This data will be input elsewhere. Now that we have our field names, we need to identify the field type values for each field. We will focus on using Python. See Table 9.3 below.
The field tag is the last element. It works as a place holder. Tags are simply a number ranging from 1 to 536, 870, 911. However, there are some rules that come with these tags:

- The number may only be used once so that it is unique (more on this later).

VALUE	DEFAULT
last_name	String (UTF-8 or 7-bit ASCII)
first_name	String (UTF-8 or 7-bit ASCII)
phone_number_cell	Int (int32)
email	String (UTF-8 or 7-bit ASCII)
unique_id	Int (int32)
date_time_created	float (could be a double in another language)
profile_picture	Bytes

Table 9.3: Field Name Python Example

- Numbers 19000 through 19999 cannot be used. Reserved by Google.

There are also some strategies to speed up the data with these tags:

- Numbers 1 to 15 use only 1 byte, so these are used for fields used most often.
- Numbers 16 – 2047 use 2 bytes

Now we put this together in the code with the Field Type, Field Name, and Field Tag.

Fig. 9.8: Code Structure

A correct schema definition for our address book example could thus look like the following:

```
message AddressBook {
    string last_name = 1;
    string first_name = 2;
    int32  phone_number_cell = 3;
    string email = 4;
    int32  unique_id = 5;
    float  date_time_created = 6;
    bytes  profile_picture = 7;
```

As you can see above the address book has 7 assigned fields. Each field is defined by a Type, Name, and unique Tag. Certain rules still apply to the message fields [27]:

- *singular*: Such fields have the cardinality 1. Thus, a message can only have none or exactly one of this field values. This is the default rule.

- *repeated*: Array or list of values. It can be repeated any number of times - even zero times.

Since the keyword *singular* is the default case, it can be omitted when defining a field. Once we have chosen the field tag number, that number is unique and cannot be reused. However, we could change the *.proto* file by commenting out a field. Field names or field tags can also be reserved for future use. Using a reserved field may cause compiler issues if the data type is not identified correctly.

For example, to define the field other phone numbers as a list, we can use the following assignment:

```
    repeated string other_phone_numbers = 8;
```

Enums

An *Enumeration (Enum)* is used when the values for a field are known or fixed. An Enum must start with tag 0 (default value). An example could be a status: Unknown Status (default), Read, Unread, Sent. The Enum values are all capitalized (upper case). See the example below. Here we added the employee's employment status. The default value is the first one tagged with zero:

```
syntax = "proto3";

/* This is Protocol Buffers
* for FORMOBILE */

message AddressBook {
    string last_name = 1;
    string first_name = 2;
    int32  phone_number_cell = 3;
    string email = 4;
    int32  unique_id = 5;
    float  date_time_created = 6;
    bytes  profile_picture = 7;
    // profile or avatar (jpg) file

    repeated string other_phone_numbers = 8;

    //Employee Status as an Enum
    enum EmployeeStatus{
        UNKNOWN_EMPLOYEE_STATUS = 0;
        CURRENT = 1;
        RETIRED = 2;
        RESIGNED = 3;
        APPLICANT = 4;
        FIRED = 5;
    }

```

```
28      /* This is an example of an ENUM notice
29      *it is ALL CAPS and the default starts with zero */
30      EmployeeStatus employee_status = 9;
31  }
```

Nesting

Messages can be added inline into another message. Nesting allows us to have message(s) types within a message type. This functionality is well known in programming languages and is called aggregation. That means some other message type is part of a second message type.

! Attention

Here a message refers to code and not a chat message.

In the example shown below, the address entry has been added to include street, city, zip code, and country. Notice the indentation. In this example, the original message refers to the `AddressBook`, and the nested message refers to the message `Address` that starts on line30. The `enum` used tag 9 and the nested `message` is now assigned tag 10. This is now defined as `AddressBook.Address`.

```
1  syntax = "proto3";
2
3  import "myproject/timestamp.proto";
4
5  message AddressBook {
6      string last_name = 1;
7      string first_name = 2;
8      int32  phone_number_cell = 3;
9      string email = 4;
10     int32  unique_id = 5;
11
12     //....
13
14     //Nesting allows us to define a message within a
15     //message (notice the indentation)
16     message Address{
17         string street_number = 1;
18         string street = 2;
19         string city = 3;
20         string zip_code = 4;
21         strong country = 5;
22     }
23     Address employee_address = 10;
24 }
```

Importing & Packages

Importing allows us to use other `.proto` file(s) or package(s) with the code you need from a different proto file. Below is a `timestamp.proto` file that has the set up for an epoch time stamp, which will be shown on the following pages.
When importing, we use *import* followed by the full path where the file is located ending with a semicolon, as seen below. Code can be compiled and put into a package. Protocol Buffers are no different, which is helpful for other coding languages. This also helps to avoid naming conflicts. A package can be created and then imported into a protocol buffer. Following is the `timestamp.proto` file. The package name is `google.protobuf.timestamp` and save the file to the same directory as the `formobileAB.proto` file.

```
syntax = "proto3";

package google.protobuf.timestamp;

option csharp_namespace = "Google.Protobuf.WellKnownTypes";
option cc_enable_arenas = true;
option go_package = "github.com/golang/protobuf/ptypes/timestamp";
option java_package = "com.google.protobuf";
option java_outer_classname = "TimestamProto";
option java_multiple_files = true;
option obj_class_prefix = "GPB";
```

```
message Timestamp {
  // Represents seconds of UTC time since Unix epoch
  // 1970-01-01T00:00:00Z. Must be from 0001-01-01T00:00:00Z to
  // 9999-12-31T23:59:59Z inclusive.
  int64 seconds = 1;
  // Non-negative fractions of a second at nanosecond resolution.
  // Negativebsecond values with fractions must still have
  // non-negative nanos values that count forward in time.
  //Must be from 0 to 999,999,999 inclusive.
  int32 nanos = 2;
}
```

To include the message definition of a Timestamp to our address book, we have to open `formobileAB.proto` file and add the imported `proto` file as well as the package name. Now we have to change the ‘date_create’d field so that the timestamp epoch time is recognized from the package:

```
1 syntax = "proto3";
2
3 /* This is Protocol Buffers
4 * for FORMOBILE */
5 import "google/protobuf/timestamp.proto";
6
```

```
7  package google.protobuf.timestamp;
8
9  message AddressBook {
10     string last_name = 1;
11     string first_name = 2;
12     int32  phone_number_cell = 3;
13     string email = 4;
14     int32  unique_id = 5;
15     google.protobuf.timestamp.Timestamp date_created = 6;
16     bytes  profile_picture = 7;
17     // profile or avatar (jpg) file
```

Now we have some idea of how a protocol buffer `.proto` file is created. The `.proto` file itself does not contain user data but just the schema. We will find such schema definitions in our analysis. However, they may appear like the `timestamp.proto` file shown in Fig. 9.9. Our analysis tool or a hex viewer may not be the best way to view this file. Here, an ordinary text editor is certainly the better choice.

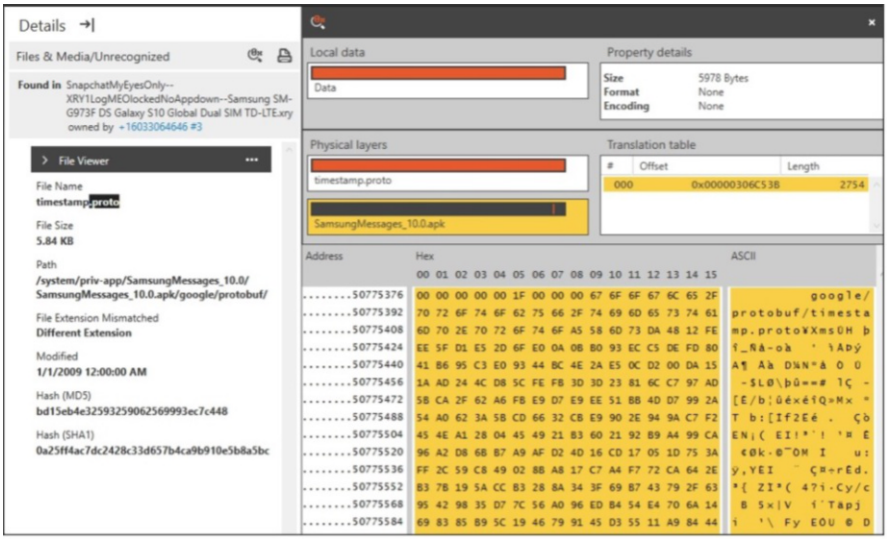


Fig. 9.9: timestamp.proto demonstrated in a Hex Viewer

The above information covers some of the code options for creating the `.proto` file. More information can be found at [27]. We will now look at the `.proto` file from a forensic analysis perspective. But first, we have to transfer our newly created message type into a concrete programming language.

9.2.2 Compiling Your Protocol Buffer

Once the custom data structures are defined as desired in the .proto file, generate the classes needed to read and write the protobuf messages. For this purpose, apply the protocol buffers compiler (protoc) to the configuration file. The `protoc.exe` is what we will be using to look at the data that we find. So first, let us see how it is used to serialize or encode the data from a protocol buffer file. The link to obtain ProtoC is `www.github.com/protocolbuffers/protobuf/releases`. ProtoC will generate code from the Proto File to the supported language. A template for coders to follow and use the defined terms.

First, we have to specify the directory to search for imports. It may be specified multiple times; directories will be searched in order. If not given, the current working directory is used. If not found in any of these directories, the `--descriptor_set_in` descriptors will be checked for required `proto` file. Next, we have to define the output language:

```
--cpp_out=OUT_DIR           Generate C++ header and source.
--csharp_out=OUT_DIR        Generate C# source file.
--java_out=OUT_DIR          Generate Java source file.
--js_out=OUT_DIR            Generate JavaScript source.
--objc_out=OUT_DIR          Generate Objective-C source.
--php_out=OUT_DIR           Generate PHP source file.
--python_out=OUT_DIR        Generate Python source file.
--ruby_out=OUT_DIR          Generate Ruby source file.
```

In this case, we will be using Python `--python_out=OUT_DIR`. Other languages like GO are supported and can be found referenced online. Now to take the formobileAB.proto file and compile the code for Python (or another language). We will place the ProtoC executable here and create a python folder.

Fig. 9.10: Python Folder Example

Then open up a command prompt in this location and follow steps 1, 2, 3 or 1, 2, 4.

1 Determine the directory name that your `proto` files are in.
2 Add Output language (Java, Python...).
3 Add Absolute path of your `proto` file with extension.
4 or all `proto` files in that location folder.

‘

Fig. 9.11: File Path Example

Analysing the Python Protobuf-Code

In our example, we have chosen Python as the target language. We will briefly discuss the file `formobileAB_pb2.proto` created in the process below. In the first section, we see imports from `google.protobuf`. One of the imports mentions reflection. This can be observed throughout the following Python example. This means the coder will have to identify the objects in their code. Descriptors are shown as well. A `serialized_pb` binary buffer could be found.

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: formobileAB.proto
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()

DESCRIPTOR = _descriptor.FileDescriptor(
  name='formobileAB.proto',
  package='',  syntax='proto3',
  serialized_options=None,
  create_key=_descriptor._internal_create_key,
  serialized_pb=b'\n\x11\x66ormobileAB.proto\"\xf7\x03\n\x0b\x41
   \x64\x64AressBook\x12\x11\n
   \tlast_name\x18\x01 \x01(\t\x12\x12\n\n
   first_name\x18\x02(\t\x12\x19\n
   \x11phone_number_cell\x18\x03 \x01(\x05\x12\r\n
   \x05\x65mail\x18\x04 \x01(\t\x12\x11\n
   \tunique_id\x18\x05 \x01(\x05\x12\x14\n
   \x0c\x64\x61te_created\x18\x06 \x01(\x02\x12\x17\n
   \x0fprofile_picture\x18\x07 \x01(\x0c\x12\x1b\n
   \x13other_phone_numbers\x18\x08 \x03(\t\x12\x34\n
   \x0f\x65mployee_status\x18\t
   \x01(\x0e\x32\x1b.AddressBook.EmployeeStatus\x12.\n
   \x10\x65mployee_address\x18\n
   \x01(\x0b\x32\x14.AddressBook.Address\x1a\x61\n
   \x07\x41\x64\x64ress\x12\x15\n
   \rstreet_number\x18\x01 \x01(\t\x12\x0e\n
   \x06street\x18\x02 \x01(\t\x12\x0c\n
   \x04\x63ity\x18\x03 \x01(\t\x12\x10\n
```

```
    \x08zip_code\x18\x04 \x01(\t\x12\x0f\n
    \x07\x63ountry\x18\x05 \x01(\t\"o\n
    \x0e\x45mployeeStatus\x12\x1b\n
    \x17UNKNOWN_EMPLOYEE_STATUS\x10\x00\x12\x0b\n
    \x07\x43URRENT\x10\x01\x12\x0b\n\x07RETIRED\x10\x02\x12\x0c\n
    \x08RESIGNED\x10\x03\x12\r\n\tAPPLICANT\x10\x04\x12\t\n
    \x05\x46IRED\x10\x05\x62\x06proto3'
)
```

Scrolling down the page we find the `AddressBook` message descriptors. You should be able to see the Field Names and the Field Tags.

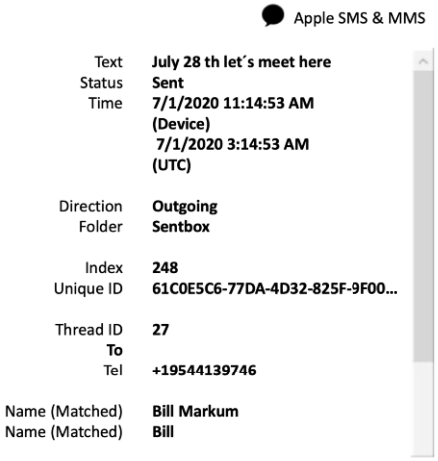
```
  full_name='AddressBook.Address',
  filename=None,
  file=DESCRIPTOR,
  containing_type=None,
  create_key=_descriptor._internal_create_key,
  fields=[
    _descriptor.FieldDescriptor(
      name='street_number',
      full_name='AddressBook.Address.street_number',
      index=0, number=1, type=9, cpp_type=9, label=1,
      has_default_value=False, default_value=b"".decode('utf-8'),
      message_type=None, enum_type=None, containing_type=None,
      is_extension=False, extension_scope=None,
      serialized_options=
      None,file=DESCRIPTOR,c
      reate_key=_descriptor._internal_create_key),
    _descriptor.FieldDescriptor(
      ↩
    name='street', full_name='AddressBook.Address.street', index=1,
      number=2, type=9, cpp_type=9, label=1,
      has_default_value=False, default_value=b"".decode('utf-8'),
      message_type=None, enum_type=None, containing_type=None,
      is_extension=False, extension_scope=None,
      serialized_options=
      None,
      ↩
    file=DESCRIPTOR,  create_key=_descriptor._internal_create_key),
```

A 2nd Example - The FormobileChat message

Having created the first example so easily, let us follow it up with a second example right away. This time it will be about defining a chat message with data fields and then generating a corresponding protobuf message. A second example was generated. The `formobilechat.proto` file has been created. After reading through this material you should have a good idea of what you are looking at. There is a message named *FormobileChat*. Followed by Field Types, Names, and Tags. There are also two `enums`

used for message direction and status. Does this data remind you of something? Chat message data, maybe?

```
syntax = "proto3";
// Formobile Protocol Buffers
message FormobileChat {
    int32 chat_thread_id = 1;
    string chat_contact = 2;
    string chat_text = 3;
    bytes chat_attachment = 4;
    float chat_latitude = 5;
    float chat_longitude = 6;
    int64 chat_timestamp = 7;
enum Chat_Direction {
    UNKNOWN_DIRECTION = 0;
    OUTGOING = 1;
    INCOMING =2;
}
Chat_Direction chat_direction = 8;
enum Chat_Status {
    UNKNOWN_STATUS = 0;
    UNREAD = 1;
    READ = 2;
}
Chat_Status chat_status = 9;
```



As with our address book example, next, we need to have the schema file compiled using the compiler protoc. The result, in our case, is again a Python source file. We could use ProtoC `-proto_path` and `-python_out` commands to generate the code for Python.

! Attention

Note there are two dashes "– " before both proto and python.

```
$> protoc --proto_path=. --python_out=.  ./formobilechat.proto
```

ProtoC took the `proto` file, output it to Python to create the `formobilechat_pb2.py` file. This file has almost 200 lines of code from a `proto` file with less than 30 lines of code.

Formobilechat_pb2.py

Even though it says `pb2`, this was made from a `proto3` file. Notice the size compared to the `.proto` file itself.

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: formobileAB.proto
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()

DESCRIPTOR = _descriptor.FileDescriptor(
  name='formobileAB.proto',
  package='',  syntax='proto3',
  serialized_options=None,
  create_key=_descriptor._internal_create_key,
  serialized_pb=b'\n\x11\x66ormobileAB.proto\"\xf7\x03\n\x0b\x41
   \x64\x64AressBook\x12\x11\n
   \tlast_name\x18\x01 \x01(\t\x12\x12\n\n
   first_name\x18\x02(\t\x12\x19\n
   \x11phone_number_cell\x18\x03 \x01(\x05\x12\r\n
   \x05\x65mail\x18\x04 \x01(\t\x12\x11\n
```

9.2.3 Creation of a Protobufs with Python

Now it is time to generate our first chat message using the Python files generated in the previous step. Therefore, a python script must be created, so this one will be named `formobilechat.py`.

First, the `formobilechat_pb2` has to be imported into the script and followed by any other imports or packages. Without the import, we would not be able to access the message types predefined. A variable is created, identifying the Fieldnames and entering data for those fields. Since *Reflection* is used, the developer must identify the fields used in the Protocol Buffer. In programming, reflection means that a programme knows its structure (introspection) and can modify it.

Program Code <formobilechat.py>

```
import formobilechat_pb2 as formobilechat_pb2

FormobileChat = formobilechat_pb2.FormobileChat()

FormobileChat.chat_thread_id = 1
FormobileChat.chat_contact_id = "Karl Agathon"
FormobileChat.chat_text = "Patrick sorry you could not make it
    tonight to get your cut of the cash. We will use you for the
    next bank. Got something for you"
FormobileChat.chat_attachment = bytes([0xFF, 0XD8, 0XFF, 0x00,
    0x10, 0x4A, 0x46, 0x49, 0x46, 0x00, 0x01, 0x01, 0x01, 0xFF,
    0xD9])
FormobileChat.chat_latitude = 5.50559
FormobileChat.chat_longtitude = -0.08956
FormobileChat.chat_timestamp = 1616182435
FormobileChat.chat_direction = 1
FormobileChat.chat_status = 2
```

In our example, a chat message is generated with a Contact named *Karl Agathon*. In addition to the actual message text, a JPEG was also added as an attachment. The message is supplemented with position information (latitude and longitude) and a timestamp. Now that the sample message is complete, we can create a real ProtoBuf message from it in the next step.

Writing the Object to a Binary File

The message is now serialized using Protobuf and saved to a binary file. For this we create a new file named *FormobileChat.bin*. Then we write the content of the messages created with Python before into the file.

Program Code

```
with open("FormobileChat.bin", "wb") as f:
    bytesAsString = FormobileChat.SerializeToString()
    f.write(bytesAsString)
```

The output file is then located in the same directory as the Python script used to create the binary.

Remember Size = Speed

Notice the size comparisons below. The first image shows the Python Script `formobilechat.py` and the `FormobileChat.bin`. This contains the complete `chat_attachment` jpg picture binary data.

Fig. 9.12: FormobileChat.bin in File Explorer

Notice the size comparison of the original proto file, the compiled `pb2.py` file, and the binary file. Note the `FormobileChat.bin` (has the full jpg picture `chat_attachment` binary data). The `FormobileChatsmall.bin` has a portion of the `chat_attachment` binary data as seen on the previous page.

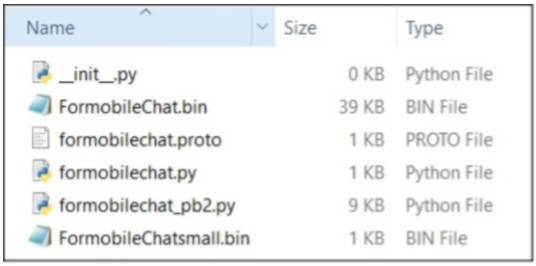


Fig. 9.13: FormobileChat.bin in File Explorer

The Raw Binary Data

Opening the file in Hex-Editor does not really do this file justice. Well we can see the chat text and the file signature of a JPG, but that is it. So how do we handle this protocol buffer data?

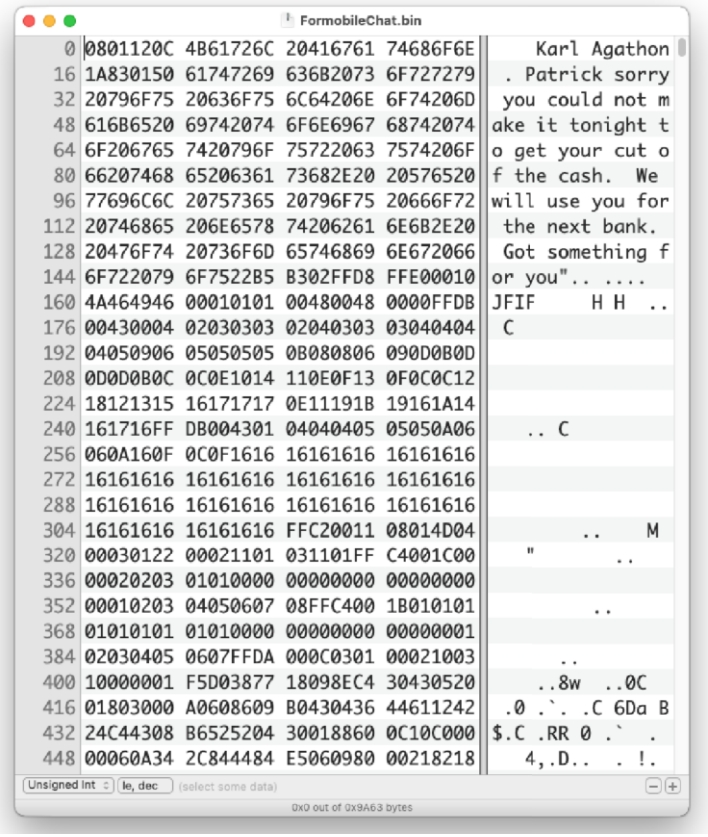


Fig. 9.14: FormobileChat.bin Hex

9.2.4 Reversing Proto Buffer Messages

In our example, we are in possession of the original *.proto* file as well as the generated binary. In practice, unfortunately, it is often the case that we do not have a schema file. But even without a interface description there is a way out.

The `protoc` compiler is not just for compiling data from a protocol buffer. But it can also be used in other ways. The `protoc` tool is very useful for showing the contents of protocol buffer data.

There is data here for us to find. We just need to know how to view it. That is where the `protoc -decode_raw` command comes in. We use the command line to decode the raw binary data from the `FormobileChatsmall.bin`. The command to use is `protoc --decode_raw < (File and Path)`. Our attempt to restore the data using the "decode raw" option was apparently only partially successful (see Fig. 9.15).

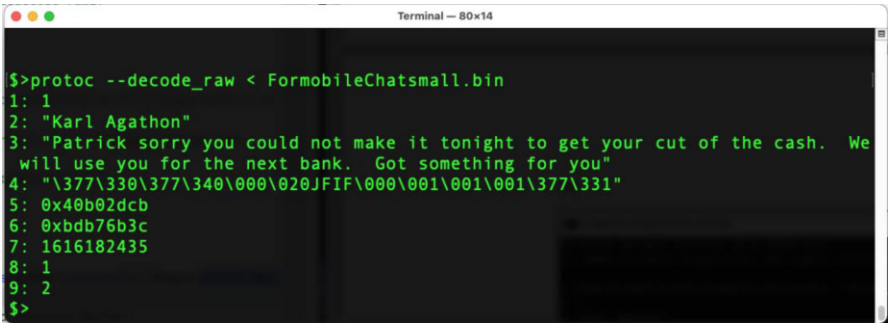


Fig. 9.15: Decoded Protocol Buffer

Fortunately, there is a solution for this as well. Thus, there are a variety of programs that provide a mostly accurate interpretation of the numerical values. The program *protobuf-inspector*[1] is one of those tools. It helps to reverse Protocol Buffers with unknown definition, i.e., missing .proto files. The command to use is `main.py < (File and Path)`

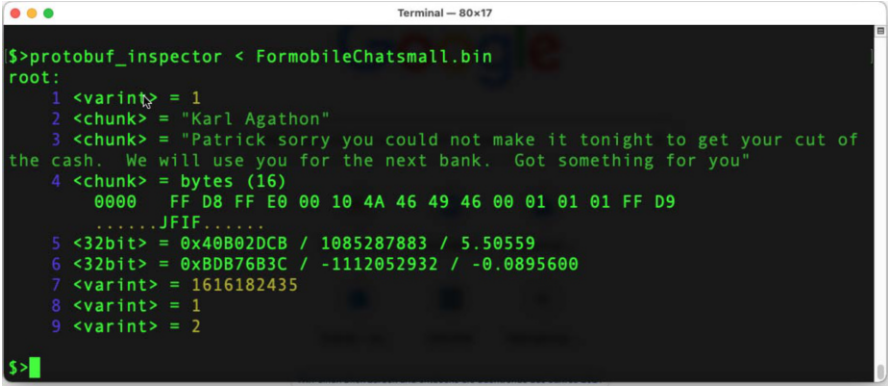


Fig. 9.16: Close up of Protobuf-Inspector decode results

With both decodes, we see the data entered, and some of it is easily understood, and other parts are not. Take note above that the `protobuf inspector` does change the octal values to hexadecimal and also translated the Longitude and Latitude in the correct Decimal 5.50559 and −0.0895600. We can compare the encoded binary message with the original Python Script results (see Fig. 9.17).

[1] www.github.com/mildsunrise/protobuf-inspector

formobilechat

```
chat_thread_id: 1
chat_contact: "Karl Agathon"
chat_text: "Patrick sorry you could not make it tonight to get your cut
chat_attachment: "\377\330\377\340\000\020JFIF\000\001\001\001\377\331"
chat_latitude: 5.50559
chat_longitude: -0.08956
chat_timestamp: 1616182435
chat_direction: OUTGOING
chat_status: READ
```

Fig. 9.17: The Original Entered Formobilechat Data (Python)

Data Conversion

While we may not know what the 1 and 2 flags mean, we can certainly look for data that we can do something about to start with. Location data in the case of 5.50559 and −0.08956 is shown in decimal, which is one way forensic tools represent it. However, what do we do when we are seeing something (maybe from a map application) that could be longitude and latitude and is in Hexadecimal: 0x40*b*02*dcb* 0x*bdb*76*b*3*c* If the Hex value starts with 8, 9, A, B, C, D, E, or F then it is a negative number. There are a few ways to do this, but the best we have been taught is HxD and a Python Script. We will show you how to use HxD's Data Inspector later in the chapter.

The Python Script requires 8 bytes, as seen below and do not name the script `struct` (as that is reserved). Also, be aware that you may input the data in the wrong spots mixing up the latitude and longitude. Test this with known data first to make sure it works in your part of the world.

Program Code

```
#convert Lat Long from hex to decimal
import struct

lat = struct.unpack('>d', b'\x40\x45\xF5\xE5\xF6\x6D\x59\x0F')[0]
long = struct.unpack('>d',b'\xc0\x51\xFA\xA3\xB1\xD3\x4B\x67')[0]

print("Latitude: ", (lat), "and the Longitude: ", (long))
```

Timestamp

In the above example, we see something that may be an epoch timestamp 1616182435. Unix time is based on the date 1970-01-01 00:00:00 (UTC), and Apple timestamps

(MAC Absolute time) use 2001-01-01 00:00:00 (UTC) as a start. Timestamps normally use seconds but may also use milliseconds, microseconds, nanoseconds etc.

The above example starts with 1 then it is probably a Unix time (when it comes to Android and Apple Devices). Apple Timestamps will most likely start with 3, 4, 5, or 6. We find epoch converter works well as an online converter or `Tempus.pyw` for an offline converter available at http://github.com/eichbaumj/Python.

Linux	CF or Mac Absolute	Apple HFS+
www.epochconverter.com	www.epochconverter.com/coredata	www.epochconverter.com/mac

Table 9.4: Epoch Timestamp Look Up Websites

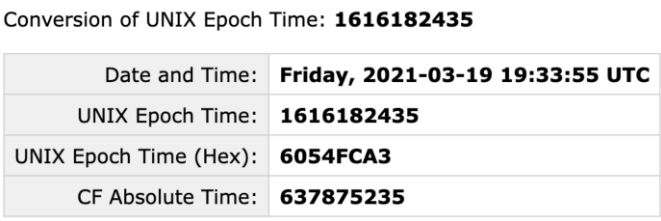


Fig. 9.18: Epoch Timestamp Look Up

Pictures or other files represented by octal data

The easiest way is to look at the data in a Hex Viewer such as HxD or your forensic tool and copy out the file. In this case, the attachment is a JPG. The file signature for a JPG file is Hex FFD8FF, and the end of the file may have Hex FFD9. You can see the start of the file below. Highlight and copy the data and save it as a .jpg type file. Example files can be found here: https://github.com/Xamnr/ProtocolBuffers.

9.3 Practical Analysis of different Proto Buffers

Analyzing digital evidence when looking thoroughly at an application can be difficult enough. Within a normal investigation of a mobile phone, the investigator already has to evaluate many different file formats. Typical file formats, which are also found in this book, are Apple Property Lists (Plists), XML, SQLite databases. Of course, you can do keyword searches for `.proto` files, but as you saw earlier, that is probably

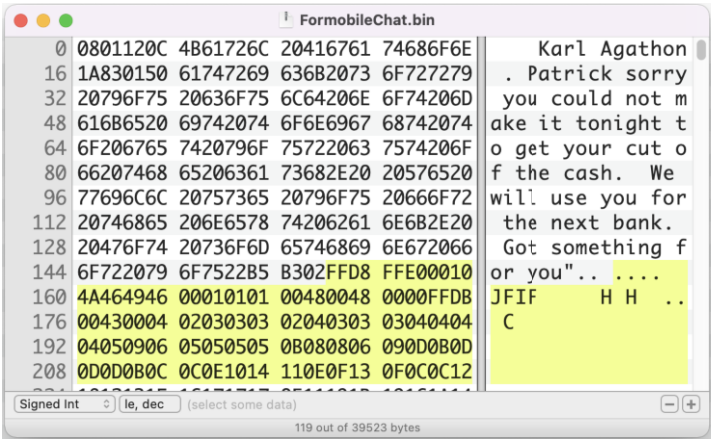


Fig. 9.19: ChatData - HexView, including file header of JPEG picture

not going to help you. To make even worser, proto buffers are often nested within other file formats rather than in their own files.

One of the issues is that these files already contain various types of data, such as Binary Large Object encoded Base 64. So, we need to be familiar with another XML or PList file within a BLOB, XML, Plist and/or Protocol Buffer Data within a BLOB. BLOBs are often also stored in a database table. In each case, we have to determine what content we are dealing with. Unfortunately, Protobuf does not have a real MagicNumber. We can only make a guess. Even more, Protocol Buffers can also be stored as GZip archive files. This adds another level of difficulty in finding these Protocol Buffers.

9.3.1 Mobile Device Artifact Examples

Some popular apps that use protocol buffers include Apple Maps, Google Maps, Badoo, Gmail, Google Allo, TamTam, Tango, WeChat, Wickr, Wire and many more. The Apple iCloud Backup system makes extensive use of protocol buffers. When dealing with application data, you are probably familiar with SQLite Databases, XML, and Apple Property List Files (.plist). You may not be aware that these files can contain data encoded Base64, such as a Binary Large Object (BLOB).

Example - Waze Navigation App

As a first example, we will use a typical app that uses Proto Buffers internally. *Waze* is a navigational guidance application for getting directions and showing the fastest

available routes. Shown on the following page is the application folder for Waze. Selected is a file named `cache_data`, shown below.

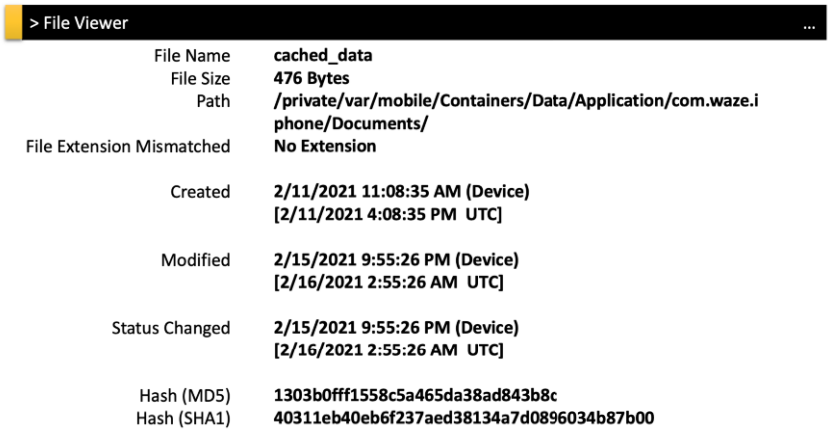


Fig. 9.20: cached_data

Let us now take a closer look at the cache file. We use the forensic tool's hex viewer to see what type of data the file contains, shown on the right. Well this is nice, we can see some data immediately (see Fig.9.21). With a simple string search we can already extract a number of artefacts. A couple things you should know about the data (see Table 9.5).

Username:	Millenium Falcon
Phone Number:	+15166618197
Home Location:	375 Main Street, New London, NH 03257

Table 9.5: Some extracted data

In fact, this example is a protocol buffer. We save the file out as a binary file adding the *.bin* file extension. Now to examine the file with both *protoc* and *protobuf inspector*. In Fig. 9.22 Waze's cached_data decoded with ProtoC. Scrolling down through the results, we come across the address and again some other data that we may or may not determine. You may not figure out what the other data items are. Again, we do not have the original code or legend. We can certainly see if data is a timestamp or location.

Below in Fig.9.23 we take the same file into `Protobuf Inspector`. As we can see, some characters on the screenshots are from escape sequences to make sure that some chars (characters) are in bold, etc. This is an easy example, in my opinion, of a protocol buffer. Using Protoc, Protobuf Inspector, or other Protocol Buffer decoding tools can help break down and show the information. In this case, the data could

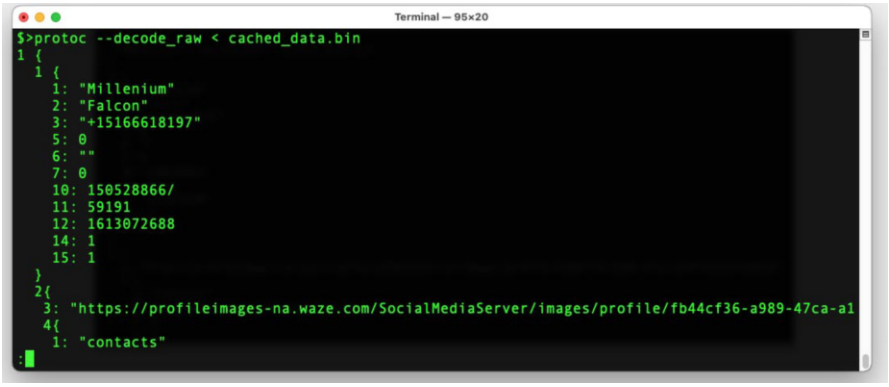


Fig. 9.21: cached_data

be seen for the most part with the hex viewer. Other instances will contain Base64 Encoded data.

BASE64 Encoding

Some of you may remember reading about BASE64 and its use with Email Attachments. This encoding scheme is to take this raw data like a picture and make sure that none of the data will cause an issue. Looking at an ASCII chart, you will notice the first 33 decimal places (0-32) are reserved for functions like BackSpace, Space, and Carriage Return.

! Attention

Please remember that Apple Property Lists, XML Files, and Databases can contain pictures or web links to pictures, and of course, Protocol Buffers. Next, we will analyze some examples of Protocol Buffers found in each.

When we send raw data, we do not want these functions to be performed. So Base64 Encoding removes these from the equation. Binary Large Objects (like a picture or even an embedded XML or Plist file) are encoded with Base64. So, what are Protocol Buffers managing? Raw data. When it comes to plists, they are usually in binary form, only rarely in text format. A forensic tool like MSAB's XAMN will show such content in a readable (XML-like) format, making binary data appear as base64. The website used in the below example to convert the raw data to Base64: www.motobit.com/util/base64-decoder-encoder.asp.

ASCII

```
 Ù  ;  Millenium
  Falcon  +15166
618197( 2 8 P £
Ì X·Ì ¨°    p x
 ¿  ghttps://pro
fileimages-na.wa
ze.com/SocialMed
iaServer/images/
profile/fb44cf36
-a989-47ca-a1f0-
b9296fb80da6"P
contacts"0d9f117
0b6404cde3300963
ba61bd64e78febf8
fc5d7f40e28dd099
d2c5580a6a8 @ (
H          "   R
Q  NS
   " *       |  Ô
÷:   ¨ ÞÀÖÝyyyyyy
' óâÛ    375 Main
 St" Main Street
* New London2 NH
: B 375J(googleP
laces.ChIJ_4k6xS
b44YkRjbQDEYOwBk
OP  ¯û Z    Home
 " (£ üÀî.  @ Hî
 ¬ð P#Z 1|global
|504040814¨
```

Fig. 9.22: cached_data

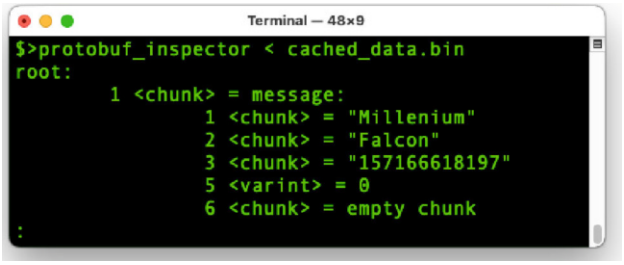


Fig. 9.23: cached_data in Protobuf Inspector

Example: Apple Web Cache file

In Fig.9.24 you can see a Apple Web Cache file. The filename is *12.xml*. The BLOB is highlighted. But what is it about in this case? Is it a protocol buffer or something else?

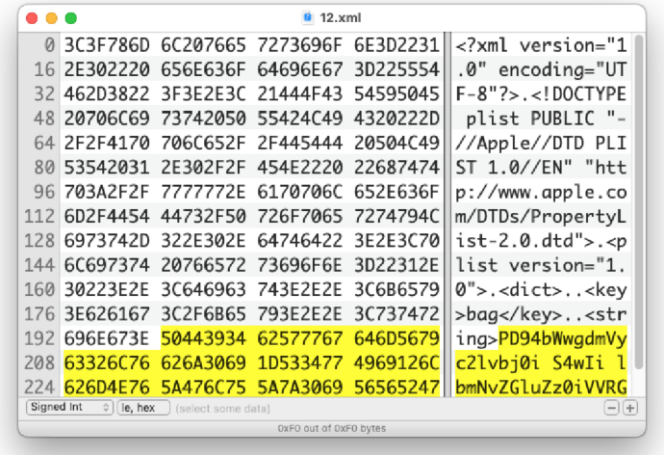


Fig. 9.24: 12.xml File Highlighting Encoded Data

We get the answer when we convert the raw data - probably BASE64 encoded - back into a normal UTF-8 string. The result is shown in Fig. 9.25. The BASE64 converted data, and we will notice that this is an XML file within an XML File. What can we learn from this? It does not always have to be a protobuf.

Source data from the Base64 string:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>refresh_cycle</key>
        <integer>86400</integer>
        <key>release</key>
        <dict>
                <key>gonzo</key>
                <dict>
                        <key>clients</key>
                        <array>
```

Fig. 9.25: 12.xml File Base 64 Converted Data

Identifying Base64 Encoded Data

You know that Binary Large Objects (BLOBS) are usually, if not always, encoded with BASE64. Sometimes you do not know. As seen below, we look for the tell-tale equals sign “=” or two equal’s signs “==” at the end of the data.

! Attention

Note the BASE64 encoded data does not always end with an equals sign.

```
CBYQACDA0QIoADAAYAKBfbWIbIcCwsJBKK5NOicKJQiL84igk82e1/
kBEhIJKzQQy2axRUARwjl2dDMFUsAYrk2QAwFA47ejkYoowAwB0gwk
OEVGRkMyRTAtQkQ1Qy00MTIGLTICQkMtMjRDRERBMzZBNDQw@WAA@=
```

This content data is then highlighted and copied. We saved it and decoded it using a base 64 decoder. In this case, we used James Eichbaum's Base64 Decoder (http://github.com/eichbaumj/Python). We then save the data as a `.bin` file and review it in the HxD Hex Editor (see Fig. 9.26). Protocol Buffer data does not have a file signature per se. If you recall, there are different Scalar values int32, int64, string, bytes, etc. . . . . Well these all have associated wire types. We have to look for hex values like 08, 09, 0A. In a protobuf, those values all correspond to key = 1 with different wire types:

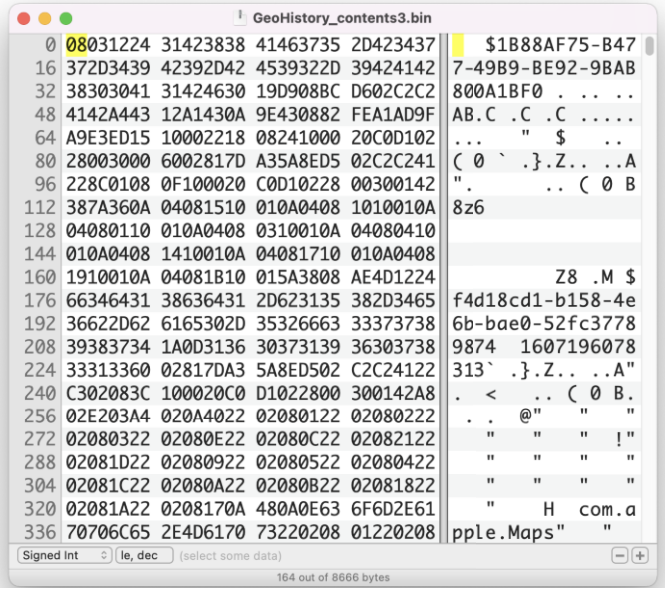


Fig. 9.26: Base64 Decode: GeoHistory Contents

08	varint (A variable length integer)
09	64 bit
0A	length delimited

Table 9.6: Typical protobuf start values

A value like 08 is not a header byte. It is just a common value protobufs start with. If interpreted as key and type it translates to key = 1, type = varint. Another common byte at the beginning of protobufs is 0x0A which translates to key = 1, type = length delimited (i.e. nested message, string, byte array). Remember: We can only make an educated guess about the binary content since protobufs directly start with the serial data stream.

> Important

A file header for a GZIP File is Hex 1F8BC8 the file will then have to be saved and unzipped.

As we know now, a protocol buffer message is a series of key-value pairs. Therefore, the serialised message consists of a series of key-value pairs that are stored one after the other in the data stream. When a message is encoded, they are concatenated into a byte stream. The binary version of a protobuf message uses the field's number as the key. A concrete name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the `.proto` file). When the message is being decoded, the parser needs to skip fields that it does not recognise. This way, new fields can be added to a message without breaking old programs that do not know about them. To this end, the "key" for each pair in a wire-format message is two values – the field number from your `.proto` file, plus a wire type that provides just enough information to find the length of the following value. Mostly, this key is referred to as a tag [23].Fig. 9.7 demonstrates the wire types available.

Type	Meaning	Usage
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Table 9.7: Available Wire Types

Just recall the *formobilechat* message from the earlier section. The protobuf binary (small one in this case) is shown in Fig. 9.27. The figure demonstrates the raw data; notice that it starts with 0x08.
The problem is we can figure out where it starts with these Hex values, possibly, but where does it end? Is it the end of the file, or only for a few bytes? Or is it 188 bytes like the aforementioned example. That is when the developers reply, "Welcome to our world."

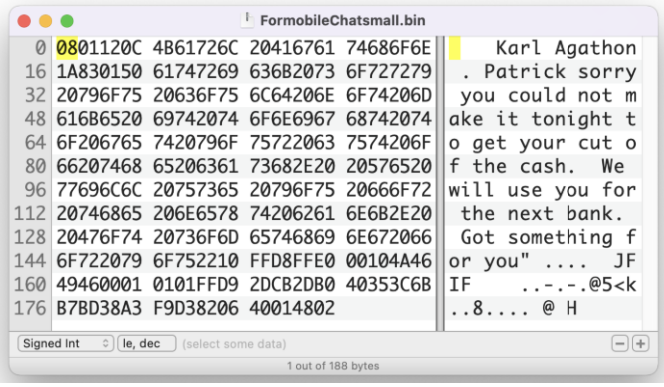


Fig. 9.27: Notice The Variable Integer 0x08 at the start

9.3.2 Yet another example: Apply Property List (PLIST) Files

Let's take a look at another example from our sample data. Fig. 9.28 shows data copied from the `GeoHistory.mapsdata.plist`. The value stored under the key "content" looks suspiciously like a BASE64 encoded value. And indeed, the data which appears to be BASE64 was copied and pasted into `www.motobit.com` decoder. The result was then copied and saved into notepad as GeoHistory_contents3.bin and opened into HxD.
Using the file `GeoHistory_contents3.bin` start with 08, but which one or ones? The file starts with 08. A search for Hex 08 results in 206 hits. See Fig. 9.29.
We manually look at each hit and the ASCII area for human-readable data that makes sense, which will not always be the case. In this first example, we take the results of the last hit, which starts at decimal offset 8579. Please copy the entire length to the end of the file and paste it into a new HxD file that we save and name with the offset (see Fig. 9.30).

Then the rest is decoding the data with Protoc and/or other tools. Moreover, try to figure out what we are looking at. Since the data belongs to a map application, we want to see if the hex values below are latitude and longitude. Maybe one of the other values is a date time stamp? See:

```
1: 0x4040cda6e5dc30e8
2: 0xc05c16c2f76aa800
```

Indeed, the values look like latitude and longitude in decimal. However, we do not want to assume that. This takes time and effort if we have to go through each search. Since the file started with hex 08, we can try `protoc` and our other tools against the entire `GeoHistory_contents3.bin` file. Keep in mind: Without the corresponding *.proto* file, we can only speculate about the meaning of the data.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>MSPHistory</key>
        <dict>
                <key>clientIdentifier</key>
                <string>0BB25E52-3840-48AE-90AF-DE804544E8E9</string>
                <key>records</key>
                <dict>
                        <key>09E098EF-EDEC-4129-B539-551AB39CB9F1</key>
                        <dict>
                                <key>contents</key>
                                <data>
                                CAESJDA5RTA5OEVGLUVERUMtNDEyOS1CNTM5LTU1MUFC
                                MzlDQjlGMRlihjz7AsLCQTJHChZQaG9lbml4IGFyaXpv
                                bmEgYXJjYWRlEgdBcml6b25hIiQpAADqMYWoQEAxAADY
                                82gYXMA5AAC4+yXZQEBBAAB2Mgj3W8BYAA==
                                </data>
                                <key>contentsTimestamp</key>
                                <data>
                                C7JeUjhASK6Qr96ARUTo6QAAAAE=
                                </data>
                                <key>modificationDate</key>
                                <date>2020-12-11T19:54:30Z</date>
                        </dict>
                        <key>17713B5A-4270-4CCE-BA34-398D761B5D1B</key>
                        <dict>
                                <key>modificationDate</key>
                                <date>2020-12-10T22:00:20Z</date>
                        </dict>
                        <key>1B88AF75-B477-49B9-BE92-9BAB800A1BF0</key>
                        <dict>
                                <key>contents</key>
                                <data>
                                CAMSJDFCODhBRjc1LUI0NzctNDlCOS1CRTkyLTlCQUI4
                                MDBBMUJGMBnZCLzWAsLCQUKkQxKhQwqeQwiC/qGtn6nj
                                7RUQACIYCCQQACDA0QIoADAAYAKBfaNajtUCwsJBIowB
                                CA8QACDA0QIoADABQjh6NgoECBUQAQoECBAQAQoECAEQ
                                AQoECAMQAQoECAQQAQoECBQQAQoECBcQAQoECBkQAQoE
                                CBsQAVo4CK5NEiRmNGQxOGNkMS1iMTU4LTRlNmItYmFl
                                MC01MmZjMzc3ODk4NzQaDTE2MDcxOTYwNzgzMTNgAoF9
                                o1qO1QLCwkEiwwIIPBAAIMDRAigAMAFCqALiA6QCCkAi
```


Fig. 9.28: GeoHistory.mapsdata.plist

9.3.3 Suggested Examination Process of a File

The idea is you create the process that works best for you. This may be completely working within your mobile forensic tool. The alternative is that you export out the file(s) of interest and review the data.

1. If unknown file type, then place the file into a Hex Viewer/Editor
2. Identify the File Signature (Research it if unknown to include possible file extensions). The data itself may be human readable as well.
3. Make sure the file has the right file extension
4. Open the file to view it natively i.e. as a database, xml, or plist file.
5. Look for Binary Large Objects (BLOB) or other raw data. Copy this raw data.
6. Decode this raw data with a Base64 Decoder and save. Devise a system to name the file and add a .bin file extension on it as a place holder.
7. Open this .bin file in a Hex Viewer/Editor
8. Identify the File Signature (as it could be an XML or Apple Property List file).
9. If this is an XML or PList File go back to Step 5. If not close the file and move on.

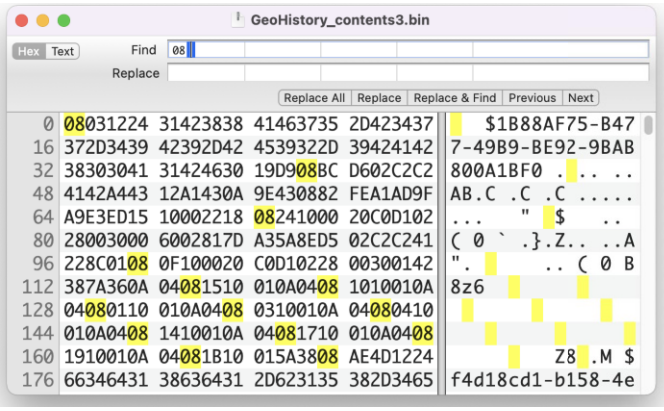


Fig. 9.29: Finding Protocol Buffer Data

Fig. 9.30: The extracted Protocol Buffer from offset 8579

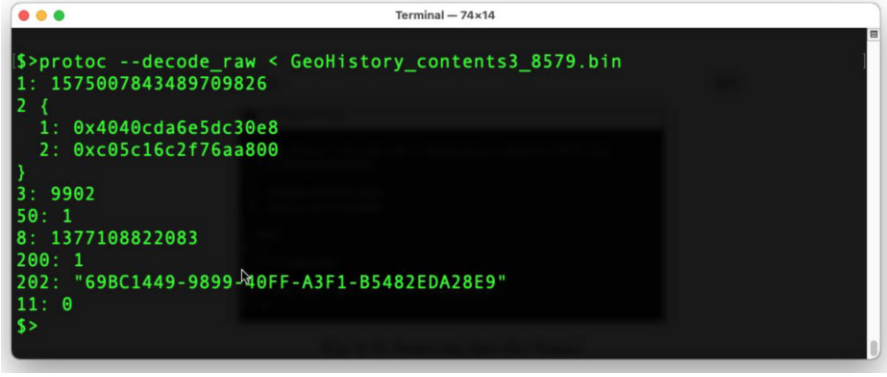


Fig. 9.31: Notice the Hex (0x) Values

10. Place the file in a folder with ProtoC executable.
11. Open a command prompt from this location
12. Type: `protoc --decode_raw < Filename.bin`
    Click enter.

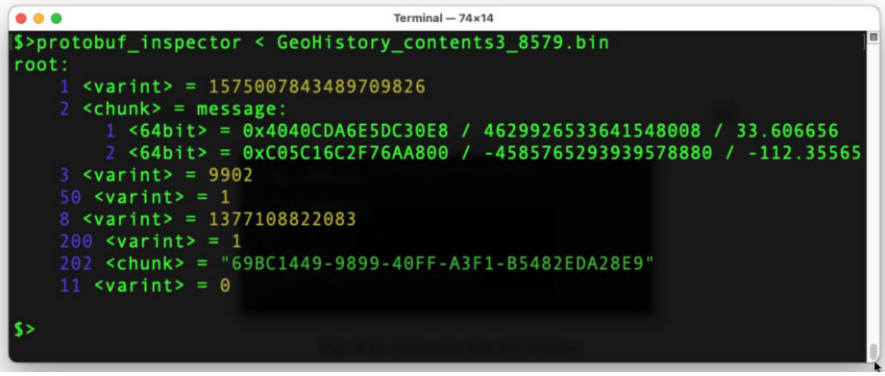


Fig. 9.32: Protobuf-Inspector converted Hex values. Lat and Long?

If you placed the file in the same directory as the *protoc* then to find the file automatically, without typing, after you type the "<" character click the space bar once and then hit tab to cycle through all of the files in the folder. Once you find the right file click enter.

13. See if you have Protocol Buffer data. If it Failed to parse the input, then that is not a protobuf (or you may need to review sections of the file for data).
14. To save the data. Highlight it and Left Click.
15. Paste into text editor.This allows you to use keywords as well.

! Attention

Now, remember the tools may obtain this data for you. However, it is nice to know where the data came from. Examining an unsupported application may have you uncovering protocol buffers for data as well.

9.3.4 Tools

We need to consider some of the tools in your toolbox for examining these artefacts. Some of these capabilities may be included with your mobile device forensic tools, such as MSAB's XRY and XAMN. If so, then these non-forensic tools will help validate your work. Most are free or have a freeware version. Some of these may cost money, so look for Freeware versions:

- Hex Viewer/Editor: HxD
- Sqlite database viewer: SQLite Expert
- XML file viewer: Notepad++
- PList file viewer: PList Editor for Windows

- Base64 Decoder: Motobit and James Eichbaum's Base64 Decoder.pyw
- Epoch Timestamp Converter: Epoch Converter Website and James Eichbaum's Tempus.pyw
- File Signature Analysis: Gary Kessler Website
- Windows Calculator in Programmer Mode
- Visual Studio
- Protocol Buffer Compiler: Proto C (protoc.exe) and Protobuf inspector

9.4 Conclusion

We have seen why Protocol Buffers are helpful. They take data make it small, and provide faster transmission speed. Coders themselves may not want to welcome the structure. As forensic examiners, we learned that understanding this structured serializing data is essential. Applications use Protocol Buffers to store data in Apple Property Lists (Plists), Binary Large Objects, and XML Files. We may find user data, time stamps, location data, and more. So, these applications alone show that protocol buffers are used and the importance of understanding them and how to analyze them.

The most important takeaway that we can provide is that now you will hopefully identify what you are looking at. Have a greater appreciation of Protocol Buffers how to make sense of this data and explain it if necessary. We learned what to look for, and now you do also.

Acknowledgements My thanks to my fellow MSAB colleagues Johan Persson, Sebastian Zankl, Oscar Choi, and Global Training Manager James Eichbaum for their time, contributions to this chapter and the forensic community.

References

1. Alendal, G., Dyrkolbotn, G. O., and Axelsson, S. (2018) Forensics acquisition — Analysis and Circumvention of Samsung Secure Boot Enforced Common Criteria Mode. Digital Investigation 24:60–67. https://doi.org/10.1016/j.diin.2018.01.008
2. Alzahrani, H. (2016) Evolution of Object-Oriented Database Systems. Global Journal of Computer Science and Technology 16(3). https://computerresearch.org/index.php/computer/article/view/1387
3. android.googlesource.com (n.d.) What is Flash-Friendly File System (F2FS). Git at Google. https://android.googlesource.com/kernel/common/+/22f837981514e157f8f9737b25ac6d7d90a14006/Documentation/filesystems/f2fs.txt. Accessed 28 May 2021
4. Apple (2020) Apple File System Reference. https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf. Accessed 28 May 2021
5. Balci, M. (2020) A Minimum Complete Tutorial of Linux ext4 File System. https://metebalci.com/blog/a-minimum-complete-tutorial-of-linux-ext4-file-system/. Accessed 28 May 2021
6. Blackberry (2015) 50 Million Vehicles and Counting: QNX Achieves New Milestone in Automotive Market. https://blackberry.qnx.com/cn/news/release/2015/6118. Accessed 28 May 2021
7. Blackberry (2010) QNX Software Systems Online Infocenter. http://www.qnx.com/developers/docs/6.5.0/index.jsp. Accessed 28 May 2021
8. Brown, N. (2012) An F2FS Teardown. Available via LWN.net. https://lwn.net/Articles/518988/. Accessed 28 May 2021
9. Caithness, A (2010) Property Lists in Digital Forensics. Available via CCL Forensics Limited. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.190.762.Accessed28May2021
10. Carrier B (2005) File Systems Forensics Analysis. Addison-Wesley Professional
11. Dewald, A. and Seufert, S. (2017) AFEIC: Advanced Forensic Ext4 inode Carving. Digital Investigation 20:83–91. https://doi.org/10.1016/j.diin.2017.01.003
12. Echessa, J. (2017) Integrating Realm Database in an Android Application. Available via AUTH0. https://auth0.com/blog/integrating-realm-database-in-an-android-application/. Accessed 28 May 2021
13. Evans, D. (2014) Log-Structured File Systems. https://www.youtube.com/watch?v=KTCkW_6zz2k. Accessed 28 May 2021
14. Fairbanks, K. D. (2012) An analysis of Ext4 for digital forensics. Digital Investigation 9:118–130. https://doi.org/10.1016/j.diin.2012.05.010
15. FPL. (2020) FlatBuffers. https://google.github.io/flatbuffers/. Accessed 28 May 2021

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0

16. Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004) Head First Design Patterns. 2nd edn. O'Reilly.
17. Garg, S. and Baliyan, N. (2021) Comparative analysis of Android and iOS from security viewpoint. Computer Science Review, 40. https://doi.org/10.1016/j.cosrev.2021.100372. Accessed 21 June 2021
18. Göbel, T. and Baier, H. (2018) Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. Digital Investigation 24:111–120. https://doi.org/10.1016/j.diin.2018.01.014
19. Google (2020) crc32c. https://github.com/google/crc32c. Accessed 28 May 2021
20. Google. (2008) google.protobuf.descriptor. https://googleapis.dev/python/protobuf/latest/google/protobuf/descriptor.html. Accessed 28 May 2021
21. Google Android Developers. (2021) Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb. Accessed 28 May 2021
22. Google Android Developers. (2021) Download Android Studio and SDK Tools. https://developer.android.com/studio. Accessed 28 May 2021
23. Google Developers. (2020) Encoding Protocol Buffers. https://developers.google.com/protocol-buffers/docs/encoding. Accessed 28 May 2021
24. Google Developers. (2020) Protocol Buffers. https://developers.google.com/protocol-buffers. Accessed 28 May 2021
25. Google Developers. (2020) Protocol Buffer Basics: C++ Protocol Buffers. https://developers.google.com/protocol-buffers/docs/cpptutorial. Accessed 28 May 2021
26. Google Developers. (2020) Protocol Buffers Language Guide. https://developers.google.com/protocol-buffers/docs/overview. Accessed 28 May 2021
27. Google Developers. (2020) Protocol Buffers Language Guide (proto3). https://developers.google.com/protocol-buffers/docs/proto3. Accessed 28 May 2021
28. Google Developers. (2020) Protocol Buffers Style Guide. https://developers.google.com/protocol-buffers/docs/style. Accessed 28 May 2021
29. gRPC Authors. (2020) About gRPC. https://grpc.io/about/. Accessed 28 May 2021
30. gRPC Authors. (2020) What is gRPC? https://grpc.io/docs/what-is-grpc/faq/. Accessed 28 May 2021
31. GSMArena (2017) Samsung Galaxy S8. https://www.gsmarena.com/samsung_galaxy_s8-8161.php. Accessed 28 May 2021
32. Haldar, S. (2015) SQLite Database System Design and Implementation. 2nd edn. https://books.google.de/books?id=OEJ1CQAAQBAJ. Accessed 28 May 2021
33. Hansen, K. H. and Toolan, F (2017) Decoding the APFS File System. Digital Investigation 22:107–132. https://doi.org/10.1016/j.diin.2017.07.003
34. Holzinger, P., Triller, S., Bartel, A., and Bodden, E. (2016) An In-Depth Study of More Than Ten Years of Java Exploitation. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York. 10.1145/2976749.2978361 p 779–790
35. Horsman, G. (2018) Framework for Reliable Experimental Design (FRED): A Research Framework to ensure the dependable Interpretation of Digital Data for Digital Forensics. Computers & Security 73:294–306. https://doi.org/10.1016/j.cose.2017.11.009
36. Hörz, M. (2020) HxD - Freeware Hex Editor and Disk Editor. https://mh-nexus.de/en/hxd/. Accessed 28 May 2021
37. Huebner, E., Bem, D., and Wee, C. K. (2006) Data Hiding in the NTFS File System. Digital Investigation 3(4):211–226. https://doi.org/10.1016/j.diin.2006.10.005
38. IBM. (2019) Relational Databases. Relational Databases Explained. https://www.ibm.com/cloud/learn/relational-databases. Accessed 28 May 2021
39. ISO/IEC (2021) ISO/IEC 9075-1:2016. Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/35/63555.html. Accessed 28 May 2021

40. Karaiskos, C (2018) Understanding Apple's Binary Property List Format. https://medium.com/@karaiskc/understanding-apples-binary-property-list-format-281e6da00dbd. Accessed28May2021
41. Kernel.org (2020) Dynamic Structures. https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html. Accessed 28 May 2021
42. Kim, J. (2012) F2FS: Introduce Flash-Friendly File System. Available via LWN.net. https://lwn.net/Articles/518718/
43. Krebs,B. (2017) Beating JSON performance with Protobuf. Available via Auth0. https://auth0.com/blog/beating-json-performance-with-protobuf/. Accessed 28 May 2021
44. Leachi, P., Mealing, M. and Salz, R. (2005) A Universally Unique Identifier (UUID) Namespace. https://www.ietf.org/rfc/rfc4122.txt. Accessed 28 May 2021
45. Lee, C., Sim, D., Hwang, JY., and Cho, S. (2015) F2FS: A New File System for Flash Storage. 13th USENIX Conference on File and Storage Technologies, Feb. 2015. https://www.usenix.org/system/files/conference/fast15/fast15-paper-lee.pdf. Accessed 28 May 2021
46. Lee, C., Sim, D., Hwang, JY., and Cho, S. (2015). F2FS: A New File System for Flash Storage. USENIX FAST 2015, Santa Clara, CA, USA. https://www.usenix.org/sites/default/files/conference/protected-files/fast15_slides_lee.pdf. Accessed 28 May 2021
47. Levin, J (2013) Mac OS X and iOS Internals. John Wiley & Sons
48. MongoDB. (2021) Realm Files. https://docs.mongodb.com/realm/sdk/node/fundamentals/realms/#realm-files. Accessed 28 May 2021
49. MongoDB. (2021) Realm Object Types and Schemas. https://docs.mongodb.com/realm/sdk/node/fundamentals/realms/#object-types---schemas. Accessed 28 May 2021
50. Merkle, R. C. (1980) Protocols for Public Key Cryptosystems. 1980 IEEE Symposium on Security and Privacy. 122–122. doi: 10.1109/SP.1980.10006
51. Nikkel, B. J (2009) Forensic Analysis of GPT Disks and GUID Partition Tables. Digital Investigation 6(1):39–47. https://doi.org/10.1016/j.diin.2009.07.001
52. Nordvik, R., Porter, K., Toolan, F., Axelsson, S. and Franke, K. (2020) Generic Metadata Time Carving. Forensic Science International: Digital Investigation 33. https://doi.org/10.1016/j.fsidi.2020.301005
53. Olsen, J. (2017) Hard Drives: How Do They Work? Techbytes. April, 2017. https://blogs.umass.edu/Techbytes/2017/04/04/hard-drives-how-do-they-work/. Accessed 28 May 2021
54. Opyrchal, L. and Prakash, A. (1999) Efficient Object Serialization in Java. Proceedings of 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware. 10.1109/ECMDD.1999.776421 p 96–101
55. Oracle Corporation (2010) Character and Block Devices. https://docs.oracle.com/cd/E19253-01/817-5789/fgoue/index.html. Accessed 28 May 2021
56. Oracle Cooperation. (2021) Object Serialization Stream Protocol. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html. Accessed 28 May 2021
57. Oracle Cooperation. (2021) Serializable Objects - The Java Tutorials. https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html. Accessed 28 May 2021
58. Panfilov, M. (2019) What the hell is protobuf? Available via MEDIUM. https://blog.usejournal.com/what-the-hell-is-protobuf-4aff084c5db4. Accessed 28 May 2021
59. Pawlaszczyk, D. and Hummert, C. (2021) Making the Invisible Visible – Techniques for Recovering Deleted SQLite Data Record. International Journal of Cyber Forensics and Advanced Threat Investigations 1(1–3):27–41. 10.46386/ijcfati.v1i1-3.17

60. Pawlaszczyk, D. and Hummert, C. (2019) "Alexa, tell me ..." - A forensic examination of the Amazon Echo Dot 3 rd Generation. International Journal of Computer Sciences and Engineering 7(11):20–29. https://doi.org/10.26438/ijcse/v7i11.2029
61. Pawlaszczyk, D. (2017) Digitaler Tatort - Sicherung und Verfolgung digitaler Spuren. In: Labudde D., Spranger M (eds) Forensik in der digitalen Welt. Springer Spektrum, Berlin, Heidelberg p 113–166
62. realm.io. (2021) realm/realm core: Core database component for the Realm Mobile Database SDKs. https://github.com/realm/realm-core. Accessed 28 May 2021
63. realm.io. (2021) realm-core/alloc_slab.cpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/alloc_slab.cpp. Accessed 28 May 2021
64. realm.io. (2021) realm-core/alloc_slab.hpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/alloc_slab.hpp. Accessed 28 May 2021
65. realm.io (2021) realm-core/alloc_slab.hpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/alloc_slab.hpp. Accessed 28 May 2021
66. realm.io (2021) realm-core/array.hpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/array.hpp. Accessed 28 May 2021
67. realm.io (2020) realm-core/node_header.hpp. Available via GITHUB. https://github.com/realm/realm-core/blob/2946c7a52449d3b8d038ff03d896b651615b8ad4/src/realm/node_header.hpp. Accessed 28 May 2021
68. realm.io (2020) realm-core/node_header.hpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/node_header.hpp. Accessed 28 May 2021
69. realm.io. (2019) realm-core/realm_dump.c. Available via GITHUB. https://github.com/realm/realm-core/blob/master/src/realm/exec/realm_dump.c. Accessed 28 May 2021
70. realm.io. (2021) realm-core/test_allocations.cpp. Available via GITHUB. https://github.com/realm/realm-core/blob/master/test/test_transactions.cpp. Accessed 28 May 2021
71. Realm.io. (2021) Realm.io. www.realm.io/. Accessed 28 May 2021
72. Realm.io. (2021) Realm Demo File Download. https://static.realm.io/downloads/realm-studio/demo-v20.realm. Accessed 28 May 2021
73. Realm.io. (2021) Realm Groups. https://github.com/realm/realm-core. Accessed 28 May 2021
74. Realm.io. (2021) Realm Studio Download. https://docs.mongodb.com/realm-legacy/products/realm-studio.html. Accessed 28 May 2021
75. Rosenblum, M. and Ousterhout, J. K. (1992) The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1):26–52. Association for Computing Machinery, New York.
76. Seacord, R. C. (2017) Java Deserialization Vulnerabilities and Mitigation. In: 2017 IEEE Cybersecurity Development (SecDev) 1:6–7. 10.1109/SecDev.2017.13
77. Science Direct. (2021) Data Definition Language. Data Definition Language - An Overview ScienceDirect Topics. https://www.sciencedirect.com/topics/computer-science/data-definition-language. Accessed 28 May 2021
78. SQLite Consortium. (2021) About SQLite. https://sqlite.org/about.html. Accessed 28 May 2021
79. SQLite Consortium (2021) Atomic Commit In SQLite - Draft. http://www.sqlite.org/draft/atomiccommit.html. Accessed 28 May 2021
80. SQLite Consortium (2021) Database File Format. https://www.sqlite.org/fileformat.html. Accessed 28 May 2021
81. SQLite Consortium (2021) Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html. Accessed 28 May 2021

82. SQLite Consortium. (2021) SQLite Database Header. https://sqlite.org/fileformat2.html. Accessed 28 May 2021
83. SQLite Consortium. (2021) SQLite Is Serverless. https://sqlite.org/serverless.html. Accessed 28 May 2021
84. SQLite Consortium (2021) Temporary Files Used By SQLite. https://sqlite.org/tempfiles.html. Accessed 28 May 2021
85. SQLite Consortium (2021) The Schema Table. https://sqlite.org/schematab.html. Accessed 28 May 2021
86. SQLite Consortium (2021) Write-Ahead Logging. https://www.sqlite.org/wal.html. Accessed 28 May 2021
87. Tanaka, K. and Saito, T. (2018) Python Deserialization Denial of Services Attacks and Their Mitigations. In: Lee, R. (ed) Computational Science/Intelligence & Applied Informatics, CSII 2018, Yonago, Japan, July 10-12, 2018. Studies in Computational Intelligence, 787:15–25. 10.1007/978-3-319-96806-3_2
88. TechTerms. (2019) NAND Definition. https://techterms.com/definition/nand. Accessed 28 May 2021
89. The Linux Kernel Organisation. (n.d.) What is Flash-Friendly File System (F2FS)? The Linux Kernel Archives. https://www.kernel.org/doc/Documentation/filesystems/f2fs.txt. Accessed 28 May 2021
90. The Sleuth Kit. (2019) HFS - SlethKit Wiki. https://wiki.sleuthkit.org/index.php?title=HFS. Accessed 17 June 2021
91. Vanura J. and Kriz P. (2018) Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats. In: Ferreira, J. and Spanoudakis, G. (eds) Services Computing – SCC 2018. SCC 2018. Lecture Notes in Computer Science, vol 10969. Springer, Cham p 413–61
92. Vickie, L. (2020) Exploiting PHP Deserialization - Intro to PHP Object Injection Vulnerabilities. Available via MEDIUM. https://medium.com/swlh/exploiting-php-deserialization-56d71f03282. Accessed 28 May 2021
93. Weyns D., Truyen E., and Verbaeten P. (2003) Serialization of Distributed Execution-State in Java. In: Aksit M., Mezini M., Unland R. (eds) Objects, Components, Architectures, Services, and Applications for a Networked World. NODe 2002. Lecture Notes in Computer Science, vol 2591. Springer Spektrum, Berlin, Heidelberg p 413–461
94. Azhar, M. A. H., Barton, T., and Islam, T. (2018). Drone Forensic Analysis Using Open Source Tools. The Journal of Digital Forensics, Security and Law. https://doi.org/10.15394/jdfsl.2018.1513
95. Mikhaylov, I. (2016). Forensic analysis of Flash-Friendly File System (F2FS). Digital Forensics Corp. https://www.digitalforensics.com/blog/forensic-analysis-of-flash-friendly-file-system-f2fs/
96. Zhai Yujia, Li Tao, and Hu Aiqun. (2020). Data forensics method of mobile terminal F2FS image file. Cyberspace Security, 11(8). http://www.css.ccidgroup.com/EN/Y2020/V11/I8/11. Accessed 28 May 2021
97. Larabel, M. (2020). ATGC Could Come In Linux 5.10 For F2FS, Much Faster Decompression Speeds Too—Phoronix. Phoronix. https://www.phoronix.com/scan.php?page=news_item&px=F2FS-ATGC-Dev-Branch. Accessed 28 May 2021
98. Yu, C. (2020). Support Age-Threshold based Garbage Collection for f2fs. LWN.Net. https://lwn.net/Articles/828027/. Accessed 28 May 2021

Index

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0