DESCRIBING DATA FORMAT EXPLOITS USING BITSTREAM SEGMENT GRAPHS

Michael Hartle[1], *Daniel Schumann*[1], *Arsene Botchak*[1], *Erik Tews*[2], *Max Mühlhäuser*[1]

Telekooperation[1] /
Kryptographie und Computeralgebra[2],
Dept. of Computer Science, TU Darmstadt / Germany

ABSTRACT

Exploits based on data processing bugs are delivered through crafted data that seems to follow a data format, yet is altered in some way to trigger a specific bug during processing, eg. in order to execute contained malicious code. Decomposing crafted data according to the purported data format and the function of its components that are not format-compliant is a step towards understanding the delivery mechanism of an exploit and fixing the vulnerable application. This paper demonstrates the use of Bitstream Segment Graphs for describing the structure of exploits on the example of the TIFF Jailbreak exploit for the Apple iPhone and iPod Touch with firmware 1.1.1.

Index Terms— Data format, bitstream, description

1. INTRODUCTION

A search in the National Vulnerability Database reveals several hundred reports on vulnerabilities that allow the execution of arbitrary code through crafted files. One of these, CVE-2006-3459, concerns multiple buffer overflows in the libtiff library prior to version 3.8.2 which allows the execution of arbitrary code during the processing of TIFF images. As this vulnerability also affected the Apple iPhone and iPod Touch products at least until the release of firmware version 1.1.2, it was used in the Apple TIFF Jailbreak exploit intended for unlocking these devices to individual developers and users that seek to extend the functionality of these devices beyond original boundaries.

A wide range of applications and data formats, be it for files or network packets, is affected by such vulnerabilities. They originate in bugs during data processing of applications that are exploited for the delivery of malicious code through crafted data. This data is composed from segments that serve a specific purpose, following a specific data format for a specific context. Examples are the segment that seems to follow the data format, but triggers the bug or the segment containing the malicious code which follows machine instruction set opcodes for the target device.

So, understanding the complete composition of crafted data is a step towards understanding the delivery mechanism employed by the exploit. By describing data, one can help to understand the exploit, identify the bug and develop a patch.

2. RELATED WORK

Surveying related work on data description, there is a need to distinguish between *describing data* and *representing data*. Describing data in general needs to cope with arbitrary forms of data representation, including aspects such as compression, encryption or multiplexing, whereas representing data has a normalizing purpose and thus can contend itself with a specific form of representation. Furthermore, one needs to distinguish between *describing a data format instance* and *describing a data format* as a possibly infinite set of data instances. In the following course of the paper, the terms data and data format instance are used interchangeably as long as no ambiguity is introduced.

Literature on data description is scarce at best and so far only an indirect subject of active research on data format descriptions, since describing a data format depends on means for describing its data instances.

2.1. Multimedia

Existing approaches for data format description mainly originate from the *Universal Media Access* context of Multimedia in widely heterogenous environments regarding bandwidth, computational power or processing capabilities. Known approaches include *Flavor* and *XFlavor* [1] for the automated generation of format-compliant software components or the *Bitstream Syntax Description Language* (BSDL) [2] including recombinations and extensions like *gBSDL* [3] and *gBFlavor* [4] for high-level multimedia content adaptation and filtering.

These approaches, although suited for their specific domains, fail as generic approaches for completely describing arbitrary data format instances in the face of real-world aspects such as zlib-compressed substructures in PNG or dependencies like length determinants in TIFF.

978-0-7695-3275-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICCGI.2008.21

IEEE computer society

2.2. Communications

Resulting from the need for interoperability in Communications, the *Abstract Syntax Notation One* (ASN.1) [5] allows the formal specification of a data model. Through the use of a standard ASN.1 codecs such as the *Packed Encoding Rules* (PER) [6] and others, a resulting data format is defined.

ASN.1 is used for specification of network protocols like H.323 for video conferencing and file formats such as X.509 for public key certificates. Its primary focus is on the specification of data models, with the corresponding data format depending on the used ASN.1 codec. For the use of ASN.1 in conjunction with legacy protocols, the *Encoding Control Notation* (ECN) [7] has been defined, yet its underlying model on data format instances is only given implicitly in a complex specification.

3. REQUIREMENTS

For a data description approach to be generally applicable in a real-world scenario, several requirements need to be fulfilled. First of all, we need to describe data with arbitrary alignment and size in order to cover everything. To explore its structure, we need to describe the composition of data from its semantically atomic components. For exploring the interplay between these components, we need to describe their relations and dependencies. These requirements declare varying degrees of completeness of a data description and can be restated as follows:

- *Width-completeness* is given if a data description can cover the bitstream and nothing but the bitstream. For general applicability, it mandates bit granularity of its descriptive means.
- *Depth-completeness* is given if a data description is width-complete and can provide for a bijective mapping between the bitstream and the set of all structured, independent literals contained within according to a data format definition. It mandates the existence of suited descriptive means for arbitrary bitstream transformations and encodings.
- *Dependency-completeness* is given if a data description is depth-complete and can cover all relations between bitstream segments as defined by a data format definition. It mandates suited descriptive means for describing arbitrary relations between properties of bitstream segments.

The previously identified problem of existing approaches on data description can be restated as a lack of suited means for depth-completeness, especially regarding block and concatenating transformations, eg. for handling zlib compression or the concatenation of interleaved audio/video bitstream fragments in multimedia containers for further processing.

4. MODEL

Extending a previous publication [8], we propose the Bitstream Segment Graph model for data description in the context of IT Security. The model is designed to be width- and depth-complete and thus intends to close the aforementioned gap regarding data description. It can be extended for dependency-completeness, which is subject of another publication.

4.1. Definitions

Without loss of generality, we assume data to be described in the form of a finite, consecutive sequence of bits, termed a bitstream. The following definitions include this term to make their scope explicit. Whenever no ambiguity is introduced, it may be omitted otherwise.

DEFINITION 4.1 (BITSTREAM SEGMENT): *A bitstream segment $v \in V$ represents a finite consecutive bit sequence $\varphi(v) \in B$, where $B = \{0,1\}^n, n \in \mathbb{N} \setminus \{0\}$ and V denotes a set of bitstream segments.*

$$\varphi : V \mapsto B$$

DEFINITION 4.2 (BITSTREAM SOURCE): *A bitstream source is a root bitstream segment $v_{Root} \in V$ with a defined $\varphi(v_{Root})$.*

A bitstream source represents a digital item which is composed according to a data format. Files, network packets or file systems on some storage medium are examples for octet-aligned bitstream sources.

DEFINITION 4.3 (BITSTREAM ENCODING): *A bitstream encoding is a tuple $e = (rel, v, l) \in R_E, v \in V, l \in L$ where R_E denotes a set of bitstream encodings and L denotes a set of literals. e specifies a bijective mapping relation $rel(\varphi(v), l)$ for a given v, abbreviated with $\phi(v) = l$.*

$$\phi : V \mapsto L$$

DEFINITION 4.4 (BITSTREAM TRANSFORMATION): *A bitstream transformation is a tuple $t = (rel, V_{in}, V_{out}, P) \in R_T$ where V_{in}, V_{out} denote totally ordered sets with $V_{in} \subset V, V_{out} \subset V, V_{in} \neq \emptyset, V_{out} \neq \emptyset, V_{in} \cap V_{out} = \emptyset$, R_T denotes a set of bitstream transformations and P denotes a set of parameters. t specifies a bijective mapping relation $rel(V_{in}, V_{out}, P)$ between V_{in} and V_{out} under application of P.*

Normalized bitstream transformations categorized by $|V_{in}| : |V_{out}|$ cardinality are the concatenating transformation of fragment segments into one composite segment ($m : 1$), a class of block transformations such as decompression or decryption ($1 : 1$) and the segmenting transformation of a structure into its separate elements ($1 : n$). Arbitrary transformations of $m : n$ cardinality can be composed through these transformations.

DEFINITION 4.5 (BITSTREAM SEGMENT GRAPH): *A bitstream segment graph (BSG) is a weakly connected, directed acyclic rooted graph $G = (V, E)$ with a set of bitstream segments V as vertices and a set of directed edges $E \subset V \times V$ connecting transformation input/output pairs of bitstream segments. It describes the composition of a bitstream source and is complete iff*

$$\forall v \in V\text{: } (\exists! t = (rel_t, V_{in}, V_{out}, P) \in R_T, v \in V_{in}) \oplus (\exists! e = (rel_e, v_e, l) \in R_E, v = v_e)$$

and partial otherwise.

DEFINITION 4.6 (TRANSFORMATION DEPENDENCY): *A transformation dependency exists if for a bitstream segment v there exists a nonempty set of bitstream segments $\varpi(v) \subset V$ with $t = (rel, V_{in}, V_{out}, P) \in R_T, v \in V_{out}$ that v depends on.*

$$\varpi : V \mapsto V^n, n \in \mathbb{N}$$

DEFINITION 4.7 (FUNCTIONAL DEPENDENCY): *A functional dependency of a bitstream segment $v \in V$ on a nonempty set of bitstream segments $V_{dep} \subset V$ with $v \notin V_{dep}$ exists if the data format defines a function f and mandates that $\phi(v) = f(V_{dep})$.*

An example of both a transformation dependency and a functional dependency is a bitstream segment which encodes the variable length of another bitstream segment. For extracting the latter from a segmentation, the value of the former is required as a parameter to the transformation. Another example of a functional dependency is a Cyclic Redundancy Code (CRC) on a set of bitstream segments, stored in another bitstream segment. Transformation and functional dependencies put constraints on possible orders of processing for bitstream segments and validity that format-compliant software components need to obey.

4.2. Composition Algorithm

Using definitions 4.1 to 4.5, we can now describe the bijective mapping between a bitstream source and its set of contained literals in a width- and depth-complete manner. The following simple algorithm constructs a BSG step-by-step. For a construction at step x, the tuple

$$(v_{Root}, V_x, V_{leaf_x}, V_{literal_x}, R_{T_x}, R_{E_x})$$

describes a designated root bitstream segment v_{Root}, a set of bitstream segments V_x, a set of leaf bitstream segments V_{leaf_x}, a set of literal bitstream segments $V_{literal_x}$, a set of bitstream transcodings R_{T_x} and a set of bitstream encodings R_{E_x}, whereas initial values are

$$\begin{aligned} V_0 &= \{v_{Root}\} \\ V_{leaf_0} &= \{v_{Root}\} \\ V_{literal_0} &= \emptyset \\ R_{T_0} &= \emptyset \\ R_{E_0} &= \emptyset \end{aligned}$$

Starting at step $x = 1$, each step either adds a transformation or an encoding. The addition of a transformation $t = (rel, V_{in}, V_{out}, P) \notin R_{T_{x-1}}, V_{in} \subseteq V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \cup V_{out} \\ V_{leaf_x} &= V_{leaf_{x-1}} \cup V_{out} \setminus V_{in} \\ V_{literal_x} &= V_{literal_{x-1}} \\ R_{T_x} &= R_{T_{x-1}} \cup \{t\} \\ R_{E_x} &= R_{E_{x-1}} \end{aligned}$$

whereas the addition of an encoding $e = (rel, v, l) \notin R_{E_{x-1}}, v \in V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \\ V_{leaf_x} &= V_{leaf_{x-1}} - n \\ V_{literal_x} &= V_{literal_{x-1}} \cup \{l\} \\ R_{T_x} &= R_{T_{x-1}} \\ R_{E_x} &= R_{E_{x-1}} \cup \{e\} \end{aligned}$$

For step y, the tuple induces a BSG $G_y = (V_y, E_y)$ where E_y is defined as follows:

$$\begin{aligned} &\forall t = (rel, V_{in}, V_{out}, P) \in R_{T_y}, \\ &\forall v_s \in V_{in}, \forall v_t \in V_{out} : e = (v_s, v_t) \in E_y \end{aligned}$$

These steps are repeated until $V_{leaf_z} = \emptyset$, where the algorithm terminates as no further addition of either transformation or encoding to leaf bitstream segments is possible. The resulting tuple induces a complete BSG. The computational tractability of this algorithm depends on the inherent computational tractability of the underlying data format.

4.3. Representation

For the representation of a BSG, bitstream segments are categorized into *types*, based on normalized transformations and encodings as shown in Table 1. To prevent a conflicting type assignment in diagrams for bitstream segments that have both the "upward" composite type and another "downward" type such as structure, an identity transformation is inserted after the composite and the "downward" type is assigned to the newly inserted bitstream segment.

Depending on their type, segments are depicted as shown in Figure 1, where *start* and *end* denote inclusive start and exclusive end bit positions relative to the preceeding bitstream

Leaf	Bitstream segment participates in Encoding	Transformation	Type
yes	no	no	Generic
yes	any	no	Primitive
no	no	segmentation input	Structure
no	no	transformation input	Transcode
no	no	concatenation input	Fragment
no	no	concatenation output	Composite

Table 1. Types of bitstream segments

segment(s), *type* denotes the bitstream segment type, *parameter* denotes a parameter for some roles and *id* denotes some plaintext identification. If all segments in a BSG are octet-aligned, then more convenient byte positions can be used for start and end, which is the case in the following BSG on the Jailbreak TIFF exploit.

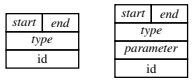

Fig. 1. Visual representations; generic, structure and composite bitstream segments (left); fragment, primitive and transcode bitstream segments (right)

5. EXAMPLE

For building a BSG instance for the Apple TIFF Jailbreak exploit, we used format-related information from the TIFF file format [9], the ARM Instruction Set [10], Darwin system calls [11] and existing ARM-based iPhone shell code in MetaSploit [12, 13]. Using the BSG model, we present a description of the exploit with focus on its functional components. Figures 2, 3, 4 and 5 show the relevant parts of the BSG instance.

5.1. TIFF file structure

The Jailbreak TIFF file is composed from the Image File Header (IFD), a so-called Strip with fake image data, an Image File Descriptor (IFD#0) and a DotRange structure.

An application starts by processing the IFD structure to obtain the file byte ordering (0x4949 for little endianess), to identify the file as TIFF document (0x2a) and to obtain a pointer to the start byte offset of the first IFD structure (0x1e/30). It then can access the IFD#0 structure, which begins with a counter on the number of IFD entries (8), followed by the IFD entry stuctures and closes with a pointer to the start byte offset of the next IFD structure, which is set to 0 to mark it as the last IFD. The IFD entry structures are composed from a tag that identify the function of their payload, a type that describes the data type stored, a count for the number of data type elements stored and the actual value, whereas for data longer than 4 bytes, not the actual data is stored, but rather a byte offset pointer into the file. As can be seen here nicely, the design choice of using pointers and length determinants in a data format can increase the susceptibility of applications for buffer overflow attacks.

According to the contents of the IFD entries #0 to #7, the TIFF file has a resolution of 8x8 pixels (#0 and #1) with colour information for each pixel stored in "chunky format" (#6). The actual compressed image data is located directly after the IFD structure at byte offset 8 (#4) as a sequence of 21 zero bytes (#5), followed by a zero padding byte to the word-aligned IFD structure.

Two of the remaining three IFD entries contain invalid values, as both the identifiers for the codec used for the compression of image data (IFD Entry #2) as well as for its photometric interpretation (IFD Entry #3) have no registered meaning and are thus unknown.

The last IFD entry #7 contains the actual exploit. The entry refers to a DotRange structure which is intended to describe the dot coverage for each ink when printing the color-separated image. Its size is constrained to either 2 or 2 * SamplesPerPixel, whereas the TIFF specification discourages the latter case of multiple dot ranges. The value of *SamplesPerPixel* defaults to 1 as it is not explicitly set, making the allowed DotRange size 2 bytes. The actual DotRange structure has size of 2048 byte values which are located at byte offset 132 in the file.

5.2. Executable payload structure

The malicious code of the exploit is based on the underlying ARM/BSD environment of the Apple iPhone and iPod Touch devices. The "DotRange" segment shown in Figure 5 is composed from three segments with only the first two having a significant function. The "Overflow" segment overwrites the stack pointer (SP) and program counter (PC) with suited values, whereas the second "Payload" segment contains a NOP sled as a landing zone and the "Loader" segment which contains a network code loader. The loader allocates executable memory via a call to mmap(). It then creates a socket, opens a TCP/IP connection to 91.121.18.102:80 and sends a HTTP GET request for a file with further executable code. It skips over the HTTP GET response headers, transfers the returned executable code into the allocated memory and finally executes it.

6. SUMMARY AND OUTLOOK

We presented the data description of the Apple TIFF Jailbreak using the Bitstream Segment Graph model and showed its applicability for documenting exploits with mingled, separate formats such as the TIFF file format and the ARM machine code.

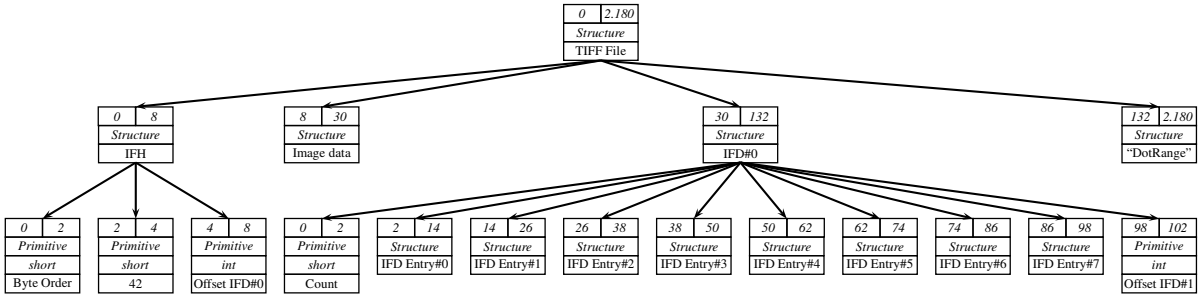

Fig. 2. Partial bitstream segment graph for the Apple TIFF Jailbreak exploit, with the DotRange structure shown in Figure 5

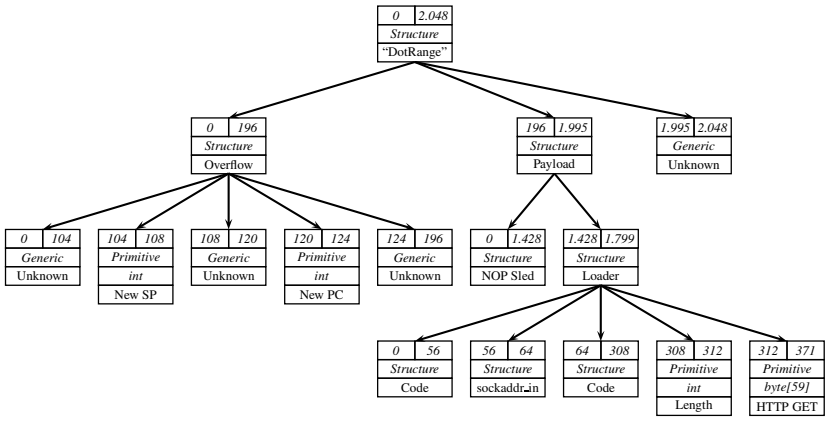

Fig. 5. Partial bitstream segment graph for the DotRange structure containing the actual exploit code

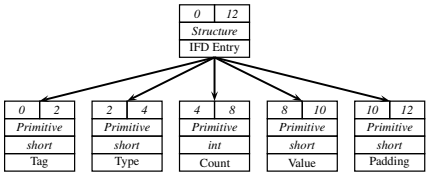

Fig. 3. Bitstream segment graph for the IFD Entries#0, #1, #2, #3 and #6, where Value represents information on ImageWidth, ImageLength, Compression, PhotometricInterpretation and PlanarConfiguration, respectively.

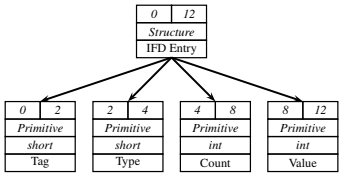

Fig. 4. Bitstream segment graph for the IFD Entry #4, #5 and #7, where Value represents information on StripOffsets, StripByteCounts and DotRange, respectively.

The BSG model provides a necessary conceptual baseline for the complete description of arbitrary data format instances. In order to extend the description to data formats as a whole, we intend it to serve as a basis for a logic-based theory on data formats, which also has applications in other domains, eg. enabling authentic long-term access to information in obsolete data formats for Digital Preservation [14].

Additional work is underway on publications for an RDF representation of BSG instances for storage and exchange and a Java-based annotation tool currently under development, with the goal of enabling use of BSGs in practice.

7. REFERENCES

[1] Alexandros Eleftheriadis and Danny Hong, "Flavor: a formal language for audio-visual object representation," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, New York, NY, USA, 2004, pp. 816–819, ACM Press.

[2] Sylvain Devillers, "An Extension of BSDL for Multimedia Bitstream Syntax Description.," in *Euro-Par*, 2003, pp. 1216–1223.

[3] Anthony Vetro, Christan Timmerer, and Sylvain Devillers, *The MPEG-21 Book*, chapter Digital Item Adaptation - Tools for Universal Multimedia Access, pp. 243–281, John Wiley and Sons Ltd, 2006.

[4] Davy Van Deursen, Wesley De Neve, Davy De Schrijver, and Rik Van de Walle, "Automatic generation of generic Bitstream Syntax Descriptions applied to H.264/AVC SVC encoded video streams," *iciap*, vol. 0, pp. 382–387, 2007.

[5] ITU-T, "Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation," ITU-T, Geneva, December 1997.

[6] ITU-T, "Recommendation X.691 (07/02) — ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)," ITU-T, Geneva, July 2002.

[7] ITU-T, "Recommendation X.692 (03/02) — ASN.1 Encoding Rules: Specification of Encoding Control Notation (ECN)," ITU-T, Geneva, March 2002.

[8] Michael Hartle, Friedrich-Daniel Möller, Slaven Travar, Benno Kröger, and Max Mühlhäuser, "Using Bitstream Segment Graphs for Complete Data Format Instance Description," in *Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT)*, 2008.

[9] Adobe Developers Association, "TIFF Revision 6.0," June 1992, last accessed 2008-03-14 at `http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf`.

[10] ARM, "ARM Documentation - Instruction Set Information," last accessed 2008-03-14 at `http://www.arm.com/documentation/Instruction_Set/index.html`.

[11] Robert Watson, "FreeBSD/Linux Kernel Cross Reference sys/bsd/sys/syscall.h," last accessed 2008-03-14 at `http://fxr.watson.org/fxr/source/bsd/sys/syscall.h?v=xnu-1228#L139`.

[12] Kevin Finisterre, "iPhone MobileSafari LibTIFF Buffer Overflow," 2006, last accessed 2008-03-14 at `http://downloads.securityfocus.com/vulnerabilities/exploits/safari_libtiff.rb`.

[13] MetaSploit Blog, "A root shell in my pocket (and maybe yours)," last accessed 2008-03-14 at `http://blog.metasploit.com/2007/09/root-shell-in-my-pocket-and-maybe-yours.html`.

[14] Seamus Ross and Margaret Hedstrom, "Preservation research and sustainable digital libraries.," *Int. J. on Digital Libraries*, vol. 5, no. 4, pp. 317–324, 2005.