Autodafé: an Act of Software Torture

Martin Vuagnoux

Swiss Federal Institute of Technology (EPFL)
Cryptography and Security Laboratory (LASEC)
http://lasecwww.epfl.ch

Abstract. Automated vulnerability searching tools have led to a dramatic increase of the rate at which such flaws are discovered. One particular searching technique is fault injection – i.e. insertion of random data into input files, buffers or protocol packets, combined with a systematic monitoring of memory violations. Even if these tools allow to uncover a lot of vulnerabilities, they are still very primitive; despite their poor efficiency, they are useful because of the very high density of such vulnerabilities in modern software. This paper presents an innovative buffer overflow uncovering technique, which uses a more thorough and reliable approach. This technique, called *fuzzing by weighting attacks with markers*, is a specialized kind of fault injection, which does not need source code or special compilation for the monitored program. As a proof of concept of the efficiency of this technique, a tool called *Autodafé* has been developed. It allows to detect automatically an impressive number of buffer overflow vulnerabilities.

keywords: fuzzer, buffer overflow, weighting attacks with markers technique, fault injection, autodafe.

1 Introduction

1.1 Buffer Overflows

When programming in a high-level language, like C, variables are declared using data type. These data types can range from integer to characters to custom user-defined structures. One reason this is necessary is to properly allocate space for each variables. C assumes that the programmer is responsible for data integrity. If this responsibility was shifted to the compiler, the resulting binaries would be significantly slower (Java, Perl, Python, C#, etc.), due to integrity checks on every variable. Also, this would remove a significant level of control from the programmer. This feature increases the programmer's control and the efficiency of the resulting programs. But it can also reduce the reliability and the security of the programs. If a programmer want to write ten bytes of data in a buffer of only two bytes of memory space, the compiler will consider this action as allowed even if it will crash the program. This is known as a *buffer overflow*, since extra bytes of data are written after the allocated space of memory.

Nowadays, the term buffer overflow has become synonymous with vulnerability or flaw, because it is sometimes possible to use buffer overflows to overwrite critical pieces of data in programs and take control of a process/computer. Poorly constructed software programs may have weaknesses such as stack overflows, heap overflows, integer overflows, off-by-one overflows, and format string bugs. For more information about buffer overflows, we recommend [9], [12], [17] and [19].

1.2 Structure of this paper

In this paper, we first explain in Section 2 how buffer overflows are currently uncovered. We will present four different techniques, with their pros and cons. Section 3 describes more precisely the technique of *fuzzing* or fault injection. Section 4 presents a more reliable approach by using a technique called *fuzzing by weighting attacks with markers*. We finally present a proof of concept of this technique, a tool called *Autodafé* which allows to automatically uncover an impressive number of buffer overflow vulnerabilities.

2 Uncovering buffer overflows

Contrary to popular belief, it is nearly impossible to determine if vulnerabilities are being identified and disclosed at an increasing or decreasing rate. According to the CERT[5] the number of disclosed vulnerabilities each year is:

2000	2001	2002	2003	2004	2005
1,090	2,437	4,129	3,784	3,780	2,874[1]

Table 1. Number of disclosed vulnerabilities per year

During the first semester of 2005, 15.75 vulnerabilities were disclosed every day. Approximately one third concerned buffer overflows. In order to uncover buffer overflows, roughly four techniques are used by automated tools. They follow chronologically the production of software programs.

2.1 Static Analysis

Automated Source Code Analysis Tools.

At the beginning, a software program is a source code. Functions in the standard C library such as `strcpy` do not perform automatically array-bound checks.

[1] This is the number of disclosed vulnerabilities during the first semester of 2005 only.

Programs using these weak functions[2] have the possibility to suffer from the buffer overflow vulnerability. By having access to the source code, an auditor is able to check if weak functions are used. Syntactic analyzers like RATS[18], Flawfinder[24] or Splint[23] can uncover these kinds of vulnerability.

In general, the current set of automated static analysis tools is lacking when it comes to uncover the relatively complicated vulnerabilities found in modern software.

Automated Binary Code Analysis Tools.

Sometimes source code is not available. Automated binary code analysis is still possible using machine language or dynamic library calls. Currently, there is no security related tool able to detect buffer overflows. Tools like Objdump[15] or APISpy32[21] give a list of the called functions, which can be unsafe.

Generally, Most of the work is done by decompilers like IDA[7] and a long and hard manual work.

2.2 Dynamic Analysis

Automated Running Analysis Tools.

If the binary code is available, executing software programs can significantly reduce the work of an auditor. Indeed running software programs give access to the reachable i.e. useful portion of code. Tools such as Valgrind[13] allow to highlight bad memory management: double freed, unfreed chunks of memory and calls of unsafe functions. Debuggers like GDB[16], Ollydbg[25], Strace[14], or Ltrace[6] are very efficient too, but they are not security related and the work is still close to a manual approach.

Fault Injection.

Fault injection or *fuzzing* is not a fully independent technique. Fault injection is normally combined with automated running analysis tools in order to simulate the use of the targeted software programs. The word fuzzing comes from fuzz[3], the first fault injection tool dedicated to uncover buffer overflows. This naive but efficient approach for finding buffer overflows is basically to supply long arguments or inputs to a program and see what happens. Fuzzers like Spike[2] and Peach[8] use a more advanced techniques.

Other tools like PROTOS[20] or Security Bug Catcher[22], much closer to fault injection than fuzzing are more complex. Using a complete specification of a protocol and an automated finite state machine describing the states of a

[2] printf, vprintf, vsprintf, wprintf, vwprintf, vswprintf, sprintf, swprintf, fprintf, fwprintf, getenv, strcat, strncat, strcpy, strncpy, stpcpy, memcpy, memccpy, bcopy, memmove, gets, system, popen, scanf, sscanf, fscanf, vfscanf, vsscanf, realpath, fgets, etc.

software program, they are able to detect if sensible states can be bypassed. Thus, this kind of tool is able to detect if an authenticated procedure can be evaded due to buffer overflows or design errors. Unfortunately, these tools must have a complete description of the protocols and the states of the audited software programs which represent a hard and long manual work.

3 Fuzzing

Black box testing with fault injection and stress testing, i.e. fuzzing, is an approach whereby an auditor uses sets of scripts designed to feed a program various inputs, different in size and structure. It is usually possible to specify how this input should be constructed and maybe how the tool should change it according to the program's behavior.

The first fuzzer fuzz[3], created in 1990 by Miller, Fredriksen and So, is basically a stream generator of random characters. It produces a continuous string of characters on its standard output file. Tested on ninety different utility programs on seven versions of UNIX, it was able to crash more than 24% of them. In 1995 Miller *et al.*[10] revisited the experiment trying to categorize the cause of these failure and compared GNU software with commercial software. In 2000 Miller and Forrester[4] tried to fuzz thirty GUI-based Windows NT applications using stream generator and random Windows messages. They were able to crash 21% of these applications.

Random testing can be considered as too trivial to be reliable. Indeed, software programs use protocols. E.g. ssh servers intend to receive at the beginning a version string:

```
SSH-1.99-openSSH_4.2
```

If the fuzzer is only able to send random characters, the probability to obtain a valid version string which passes the check is close to zero. In a paper[1] published in 2002, Dave Aitel describes an effective method, called Block-Based Protocol Analysis, implemented in Spike[2], the most used fuzzer. Protocols can be decomposed into length fields and data fields: consider the case where an auditor wants to send a long character string to a web server using HTTP. It is not enough to simply modify a captured request by replacing a variable with a longer string. The auditor must also update the *Content-Length* field in the HTTP header (POST). Block-Based Protocol Analysis allows the auditor to create blocks of data binded to length variables. Thus if the size of a block of data is modified due to string substitutions, the fuzzer is able to recalculate the correct size and send a request with a correct length value[3]. Moreover, this technique permits a complete description of protocols, delimiting fixed strings and variables.

With a block-based description of protocols, fuzzers can drastically reduce the size of the potential space of inputs.

[3] Sending a request with a wrong size can highlight vulnerabilities called *Integer overflows*. Fuzzers should be able to test this kind of flaw too.

Potential Space of Inputs

The cardinality of the potential space of inputs defines the complexity of fault injectors: fuzzers basically substitute variables for smaller, bigger and malformed strings or values. By using a random character string generator, Fuzz[3] owns an infinite potential space of inputs. In order to reduce the complexity, most advanced fuzzers combine three techniques:

Partial description of protocols. In order to omit useless tests. See ssh example above.

Block-based protocols analysis. This technique permits to recalculate length fields after substituting data. See HTTP example above.

Library of substituted strings or values. Substituting variables with random values is irrelevant. Using a library of finite substituted strings or values drastically reduces the size of the potential space of inputs. E.g. in order to highlight *format string bugs*, only a few character strings are tested, containing all the interpreted sequences[4]

Thus, the complexity of a fuzz test can be defined by the number of substitution:

$$Complexity = LF$$

Where L is the number of substituted strings or values contained in the library and F is the number of fuzzed variables. Spike uses a library with 681 entries in version 2.9 but for deep analysis, the number of entries in the library of substituted strings or values should be much bigger (dozens of thousands).

Although these optimizations increase the efficiency of fuzzers, the size of the potential space of inputs still needlessly wide. In order to affect the complexity, both L and F can be reduced. Bringing down the size of the library of substituted strings or values L is not relevant. Each entries in L is based on the discovery of a buffer overflow. Omitting entries can make the fuzzer less effective. However, limiting or arrange the fuzzed variables F could dramatically reduce the complexity.

4 Weighting Attacks with Markers Technique

This technique is used to reduce the complexity of fuzzers. If L is composed of dozens of thousands entries, removing one input in F is profitable. So, if an auditor has access to the software program (grey-boxing), he can use tracers or debuggers to obtain more information for his fuzzing.

Definition 1 (Tracer). *A tracer is a debugger able to list every dynamically called function of a software program.*

[4] Interpreted sequences are used by libc functions like printf to describe printable arguments. For example, strings are defined by "%s" and integers by "%d".

By combining runtime analysis tool with fuzzer, it is possible to know when unsafe functions like `strcpy` are used. Moreover if a vulnerability is discovered, debug functions can be used to detail the cause.

Using unsafe functions is critical when theirs arguments can be modified or controlled by users.

Definition 2 (Marker). *Every character string or value controlled by users (locally or remotely) is considered as a marker.*

Procedure

Basically, the paradigm is to automatically analyse how markers are used by the targeted software program. If a marker i.e. a controlled string or value, is used as an argument by an unsafe function, a weight related to this marker increases. When all markers are tested and weighted, the fuzzer will test firstly the markers with larger weight. Thus the cases where the probability to find a buffer overflow is superior are tested in the first place.

1. A partial description of the audited protocol is given to the fuzzer using a block-based protocol language. Every canonical element is considered as a marker. This language is defined to fragment the protocol in length fields and data fields.
2. The fuzzer uses this description to simulate a normal (previously captured and unmodified) communication with the targeted software program.
3. According to the block-based protocol language, some canonical elements are tagged as markers. Thus the fuzzer send to the tracer the list of markers it needs to watch.
4. The tracer starts the targeted software program and puts breakpoints to every unsafe functions.
5. The tracer analyses the execution of the targeted software program in order to detect if unsafe functions use markers.
6. If a marker is used by unsafe functions, the tracer gives a bigger weight to the marker and communicates its results to the fuzzer.
7. According to the weight of markers, the fuzzer classify which markers should be tested first. Markers which do not use vulnerable functions are not fuzzed during the first pass.
8. Each weighted marker is substituted to every entry in the library, in order to give rise to a buffer overflow.
9. If a fuzzed variable causes a buffer overflow, the tracer gives to the auditor additional information about the state of the targeted software program.

By reducing the cardinality of F – the fuzzed variable space – and by ordering it, the complexity is drastically declined.

5 Autodafé

In this section, we present an implementation of the fuzzing by weighting attacks with markers technique.

5.1 Block-Based Protocol Description Language

The block-based language used to describe protocols contains these functions:

```
string("dummy");                /* define a constant string */
string_uni("dummy");            /* define a constant unicode string */
hex(0x0a 0a \x0a);              /* define a hexadecimal constant value */
block_begin("block");           /* define the beginning of a block */
block_end("block");             /* define the end of a block */
block_size_b32("block");        /* 32 bits big-endian size of a block */
block_size_l32("block");        /* 32 bits little-endian size of a block */
block_size_b16("block");        /* 16 bits big-endian size of a block */
block_size_l16("block");        /* 16 bits little-endian size of a block */
block_size_8("block");          /* 8 bits size of a block */
block_size_8("block");          /* 8 bits size of a block */
block_size_hex_string("block"); /* hexadecimal string size of a block */
block_size_dec_string("block"); /* decimal string size of a block */
block_crc32_b("block");         /* crc32 of a block in big-endian */
block_crc32_l("block");         /* crc32 of a block in little-endian */
send("block");                  /* send the block */
recv("block");                  /* receive the block */
fuzz_string("dummy");           /* fuzz the string "dummy" */
fuzz_string_uni("dummy");       /* fuzz the unicode string "dummy" */
fuzz_hex(0xff ff \xff);         /* fuzz the hexadecimal value */
```

With these functions it is possible to reproduce almost every binary-based or string-based protocol. Writing protocol descriptions is a hard manual task. In order to help auditors *Autodafé* uses a tool called `adc` which verifies the syntax of the script and convert correct files into a specific format. Moreover, the Ethereal[11] protocol recognition engine is used by the tool `pdml2ad` to automatically recover 530 existing protocols by capturing legitimate communications: E.g. this is the automatically recovered description of the first packet sent during a ssh connection using the autodafe language script:

```
block_begin("packet_1");
   // name    : ssh.protocol
   // showname: Protocol: SSH-1.99-OpenSSH_4.2\n
   // show    : SSH-1.99-OpenSSH_4.2\x0a
   // size    : 21 (0x15)
   string("SSH-1.99-OpenSSH_4.2");
   hex(0a); /* \n */
block_end("packet_1");
recv("packet_1");
```

The autodafe language script is not only able to describe network based protocols but also file formats like pdf, gif, jpeg, bmp, MS-Word, Postscript, etc.

5.2 The fuzzer

The fuzzer engine, called `autodafe`, is connected to a tracer, called `adbg`, by using a TCP connection. Together, they can send fuzzed variables according to

a modifiable substituted strings and values library and implement the technique of fuzzing by weighting attacks with markers. Both Microsoft Windows based and Unix based versions of the tracer are available.

5.3 Results

By using this technique we were able to uncover eighty known buffer overflows and 865 new unreleased buffer overflows in modern software.

6 Conclusion

In this paper we have presented techniques used to uncover automatically buffer overflow vulnerabilities. In particular we have detailed fault injection or fuzzing – i.e. insertion of malformed data into input files, buffer or protocol packets. We have defined the complexity of these automated tools and the most advanced techniques used to reduce it. We have presented an innovative buffer overflow uncovering technique, called *fuzzing by weighting attacks with markers*, which uses a more thorough and reliable approach, by combining a runtime analysis tool with a fuzzer. As a proof of concept of the efficiency of this technique, a tool called *Autodafé* has been developed which is able to detect an impressive number of buffer overflow vulnerabilities.

7 Acknowledgments

A part of this work was done at the University of Cambridge: we thank all the Security Group of the Computer Laboratory of the University of Cambridge, especially Markus G. Kuhn and Steven J. Murdoch. We would also thank Serge Vaudenay and Philippe Oechslin from the Swiss Federal Institute of Technology (EPFL) - LASEC.

References

1. Dave Aitel. The advantages of block-based protocol analysis for security testing, 2002.
   http://www.immunitysec.com/resources-papers.shtml.
2. Dave Aitel. Spike, 2003.
   http://www.immunitysec.com/resources-freesoftware.shtml.
3. Bryan So Barton P. Miller, Lars Fredriksen. An empirical study of the reliability of unix utilities, 1990.
   http://www.cs.wisc.edu/~bart/fuzz/fuzz.html.
4. Justin E. Forrester Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing, 2000.
   http://www.cs.wisc.edu/~bart/fuzz/fuzz.html.
5. CERT. Cert/cc statistics 1998-2005, 2005.
   http://www.cert.org/stats.
6. Juan Cespedes. Ltrace, 2005.
   http://packages.debian.org/unstable/utils/ltrace.html.
7. Datarescue. Ida, 2005.
   http://www.datarescue.com.
8. Michael Eddington. Peach, 2005.
   http://www.ioactive.com/v1.5/tools/index.php.
9. Jon Erickson. *Hacking: The Art of Exploitation*. No Starch Press, 2003.
10. Barton P. Miller et al. Fuzz revisited: A re-examination of the reliability of unix utilities and services, 1995.
    http://www.cs.wisc.edu/~bart/fuzz/fuzz.html.
11. Gerald Combs et al. Ethereal - the world's most popular network protocol analyzer, 2005.
    http://www.ethereal.com.
12. Jack Koziol et al. *The Shellcoder's Handbook*. John Wiley & Sons, 2004.
13. Julian Seward et al. Valgrind, 2005.
    http://www.valgrind.org.
14. Paul Kranenburg et al. Strace, 2003.
    http://www.liacs.nl/~wichert/strace/.
15. GNU. Objdump, 2004.
    http://www.gnu.org/software/binutils/.
16. GNU. Gdb, 2005.
    http://www.gnu.org/software/gdb/gdb.html.
17. Gary McGraw Greg Hoglund. *Exploiting Software: How to Break Code*. Addison-Wesley Professional, 2004.
18. Secure Software Inc. Rats - rough auditing tool for security, 2002.
    http://www.securesoftware.com/rats/rats-2.1.tar.gz.
19. Nish Balla James C. Foster, Vitaly Ospinov. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Syngress, 2005.
20. Rauli Kaksonen. Protos - security testing of protocol implementations, 2002.
    http://www.ee.oulu.fi/research/ouspg/protos/.
21. Yariv Kaplan. Apispy32, 2002.
    http://www.internals.com/utilities_main.htm.
22. Philippe Oechslin. Security bug catcher, 2004.
    http://lasecwww.epfl.ch/~oechslin/projects/bugcatcher/.

23. Department of Computer Science Secure Programming Group, University of Virginia. Splint - a tool for statically checking c programs, 2003.
http://www.splint.org.
24. David A. Wheeler. Flawfinder, 2004.
http://www.dwheeler.com/flawfinder/.
25. Oleh Yuschuk. Ollydbg, 2005.
http://www.ollydbg.de/.