Flayer: Exposing Application Internals *

Will Drewry *and* Tavis Ormandy
Google, Inc.
{*wad,taviso*}*@google.com*

Abstract

Flayer is a tool for dynamically exposing application innards for security testing and analysis. It is implemented on the dynamic binary instrumentation framework *Valgrind* [17] and its memory error detection plug-in, *Memcheck* [21]. This paper focuses on the implementation of Flayer, its supporting libraries, and their application to software security.

Flayer provides tainted, or marked, data flow analysis and instrumentation mechanisms for arbitrarily altering that flow. Flayer improves upon prior taint tracing tools with bit-precision. Taint propagation calculations are performed for each value-creating memory or register operation. These calculations are embedded in the target application's running code using dynamic instrumentation. The same technique has been employed to allow the user to control the outcome of conditional jumps and step over function calls.

Flayer's functionality provides a robust foundation for the implementation of security tools and techniques. In particular, this paper presents an effective fault injection testing technique and an automation library, *LibFlayer*. Alongside these contributions, it explores techniques for vulnerability patch analysis and guided source code auditing.

Flayer finds errors in real software. In the past year, its use has yielded the expedient discovery of flaws in security critical software including OpenSSH and OpenSSL.

1 Introduction

Vulnerabilities often lay undiscovered in software due to the complexity of the code paths leading to them. Recent tools attempt to understand these paths and modify running application code, detecting flaws ranging from undefined memory use [21] to signedness conversion errors [15] to unbounded memory access [32]. In addition, symbolic evaluation and analysis frameworks, like *EXE* [8] and *SAGE* [12], and other multiple execution path analysis tools [16], have begun to augment this effort through the automated generation of dangerous input. While execution path, or flow, analysis techniques have been in use for over three decades [7], practical analysis tools for white box testing and auditing scenarios have only recently become commonplace [15] [12] [8] [32] [19].

This paper presents *Flayer*, an execution flow analysis and modification tool, and a complementary *fuzz testing* [14] technique. Flayer is implemented as a plug-in to the dynamic binary instrumentation framework *Valgrind* [17] using core functionality from its memory error detection plug-in, *Memcheck* [21]. It traces the flow of tainted, or marked, input data through an application during execution and logs the traversal of conditional jumps and system calls. Recent works, such as *autodafé* [32] and *Byakugan* [19], also rely on understanding input flow through a process. However, these tools use input pattern matching techniques for taint tracing which lack the accuracy of Flayer's dynamic binary instrumentation based approach. Flayer improves on existing taint tracing software, like *TaintCheck* [18] and *Catchconv* [15], through the addition of bit-precise taint propagation. This precision allows for taintedness to propagate into bitfields and bit arrays creating a more accurate view of the impact input has on an application's execution. Furthermore, Flayer is not solely a taint tracing tool. It also provides the ability to redirect the flow irrespective of input. Flayer can instrument the outcome of conditional jumps and function calls in the execution path based on user-supplied arguments. In addition, a library for automated execution and output processing, *LibFlayer*, is available for use along with an interactive shell interface, *FlayerSh*, for easy human interaction.

The application of Flayer's flow tracing and alteration

*First presented at the WOOT'07 First USENIX Workshop on Offensive Technologies.

functionality, *flaying*, provides a means to directly expose code obscured behind complex code paths for direct testing. This approach combined with random fuzz testing results in a lightweight, yet effective testing technique.

1.1 Paper structure

The remainder of this paper discusses Flayer, its implementation and applications. Section 2 covers the detailed implementation of Flayer. Section 3 introduces a new fuzz testing technique. Section 4 discusses other techniques enabled through the use of Flayer and its supporting libraries. Section 5 provides real world experiences where the presented software and techniques have successfully discovered security-related application flaws. Section 6 details the possibilities for future work, and Section 7 gives the conclusions drawn.

2 Flayer

2.1 Foundation

Flayer is implemented as a plug-in to Valgrind, a framework for instrumenting machine code at runtime. In particular, it is based upon functionality from Memcheck. Memcheck is a Valgrind plug-in that provides four types of memory error detection: byte-level addressability, heap allocations, memory block argument overlapping, and definedness checking. Of these, definedness checking was the basis for Flayer's taint propagation feature. Other functionality provided directly by Valgrind was leveraged for implementing taint sources and control flow alteration. In addition, Valgrind's default error output and robust command line argument handling mechanisms enabled easy automation with a simple wrapper library, LibFlayer.

2.2 Bit-precision taint tracing

Tainting is the process of tagging data with metadata that is propagated when that data is involved in a value-creating operation. The implementation of bit-precision taint tracing may be divided into three logical pieces: initial taint assignment, taint propagation and notification, and taint removal.

Taint is assigned to data based on the data sources specified on the command line. The following sources are supported: network, file, and stdin. All data originating from the network, the file system, or standard input are tainted through the instrumentation of system calls made by the target application. In most cases, this is handled by the `read` system call. As data enters the application via this kernel interface, the instrumented call checks if the source file descriptor is tainted and appropriately marks the destination memory addresses. In addition, `recvmsg` and `recvfrom` are instrumented in the same manner. File descriptor-based tainting is managed in two ways. If standard input tainting is specified, data originating from file descriptor `0` is tainted. For network and file tainting, file descriptor tracking is handled through the instrumentation of the following system calls: `open`, `socket`, `connect`, `accept`, `socketpair`, and `close`. When the data sourced from the file system is to be tainted, `open` controls whether a file descriptor is marked as providing tainted data. By default, if file tainting is enabled, all file descriptors opened with `open` will be marked. When a file descriptor is closed with `close`, it is unmarked as providing tainted data. However, tainting all input from open file descriptors may taint a large amount of data as shared libraries are loaded and files are read by the target application. The command line argument `--file-filter` exists to mitigate this problem. The argument takes a string which specifies a path prefix to the desired file, or files, to be tainted. This allows for targeted tainting of file input data. Unfortunately, there are no such filters for network tainting. If enabled, all network file descriptors are assumed to produce tainted data. Usually, this is not a burden given that network operations are not fundamental to process initialization. Along with system call instrumentation, taint may be assigned through one other mechanism: client calls. Valgrind provides a mechanism where special machine instructions may be inserted into an application, or library, at compile time through the use of C macros. Usually used from preloaded shared objects, these client calls may taint, untaint, or examine chunks of application memory.

The propagation of *taintedness*, whether data is tainted or not, is largely implemented using the undefinedness propagation technique implemented in Memcheck. In this technique, all bits in memory and registers have associated bits of metadata, shadow bits, which track taintedness. Furthermore, each value-creating memory operation has a shadow operation which calculates the taintedness of the result. This direct memory propagation approach performs the majority of the taintedness propagation. Flayer also implements an indirect technique to further expand coverage. Flayer preloads a shared library that replaces several functions in the target application which operate on strings and raw memory: `strnlen`, `strlen`, `strncmp`, `strcmp`, `memcmp`, and `bcmp`. In practice, these functions operate on memory that may be tainted but will not propagate taintedness to their return value because that value is not the direct result of a memory operation. For example, `x = y + 1` results in `x` being tainted if `y` is tainted. However, in the following example `len` will not be tainted even if `s` is:

```
char *c = s; size_t len = 0;
for( ; *c; c++ ) { len++; }
return len;
```

While it is clear to a human that the final value stored in `len` is based completely on the contents of `s`, direct memory-to-memory propagation cannot address the situation. To work around this, the replacement functions listed make use of client calls to determine if the source memory is tainted and taint the return value appropriately. If these functions have been inlined, or custom equivalents are used, the preloaded versions will not be used and taintedness will not propagate indirectly.

Taintedness propagation functions generate external notification messages. Given that Memcheck already reports on traversed conditional jumps, system call argument usage, memory access, and SIMD or FP register memory loads, Flayer inherited output that is sufficiently rich without the addition of further messages.

Memory must be untainted when it no longer contains a tainted value to avoid false positives. In most cases, memory is untainted through the taint propagation code. If an untainted value is written directly to a tainted memory location, that location will become untainted. Memory is also untainted when it is allocated or freed on the heap through `malloc`/`free` wrapper functions. All other cases are handled through Valgrind callbacks: stack creation, stack destruction, and client calls.

2.3 Execution path alteration

Flayer alters a target program's execution path through direct instrumentation of its machine code, a practice classically used in software cracking. In particular, two types of alterations are possible: forcing conditional jumps and stepping over function calls. The instrumentation occurs after machine code is translated to Valgrind's intermediate representation (IR) and before it is translated back to machine code.

Conditional jump alteration is controlled by the `--alter-branch` command line argument. This argument takes a comma-separated list of instruction pointer and value pairs joined by colons, e.g. `--alter-branch=0x8080:1,0x9090:0`. The value specified after the instruction pointer is that of the guard of the conditional jump. A value of `0` indicates that the branch should not be followed while a value of `1` will result in the branch being followed. This behavior occurs irrespective of the values involved in the conditional itself. Any conditional jump may be altered using this technique regardless of whether it is visible during taint analysis.

In addition to forcing conditional jump outcomes, Flayer allows function calls to be stepped over using the `--alter-fn` command line argument. This argument takes a similar format to `--alter-branch` except that the value may be any 32-bit integer. The address supplied is not that of the function to be skipped, but instead, the address where the function is called. At this address, Flayer adds two instructions. The first sets the value of the EAX register to the 32-bit value supplied in the command line argument. The second is a jump to the next physical instruction after the call site. This forces the function call to be bypassed while still providing a controllable return value.

2.4 LibFlayer

LibFlayer is a Python library which provides a programmatic interface to Flayer. It is comprised of several components, the most important of which is the Flayer class.

The Flayer class is the core interface of the library. It supplies the getters and setters for managing Flayer command line arguments and provides interfaces for interacting with parsed output. Through these interfaces it is possible to specify what input type to taint, what file paths to filter, and what conditional jump addresses to modify. The interface can be used directly or wrapped further for higher levels of abstraction. One such wrapper provides the interactive shell interface used by FlayerSh. In addition, some effort has been invested in the automated exploration of execution path trees using LibFlayer.

3 A new fuzz testing technique

3.1 Background

Random fault injection-based testing, or fuzz testing, is the technique of supplying random input to an application with the intent of discovering an unseen, and potentially dangerous, code path. Traditional fuzz testing is often underutilized due to its inherent limitations. In particular, exhaustive testing of an application's input space quickly becomes infeasible. Fuzz testing one or two bytes may not be prohibitive, but testing even a small set of 500 bytes requires 2^{8*500} combinations to completely exercise the input space.

While there are many specialized techniques to mitigate this exponential explosion of combinations, two generalized practices have arisen. The first is block-based [4], or format aware, fuzz testing. *Spike* [5], *PROTOS* [20], and *Peach* [11], among others, use this approach to limit the randomness in the data to just the mutation of format-specific components. This approach has shown its efficacy [4] but requires a substantial initial investment in the form of extensive format specification. Even in systems where this specification is generated automatically [32] [6], fuzz testing based on a

protocol definition may not exercise code from undocumented features or proprietary vendor extensions and may waste significant resources testing unimplemented specification features. For example, consider testing a HTTP server. WebDAV [13] alone adds nine new HTTP methods in addition to multiple new HTTP headers. The combination of these HTTP methods, headers, and their arguments takes a substantial time to explore regardless of whether the server supports the functionality.

The second technique is exemplified in the work by Vuagnoux called autodafé [32], as well as Pusscat's Byakugan [19]. The approach focuses on the use of recognizable patterns in the input stream which are detected through function hijacking or frequent memory scanning. This technique is useful for detecting which pieces of input reach specific locations, but it is limited by design. Not only is it possible for the marker text to be modified beyond recognition during execution, but the method itself introduces uncertainties in measurement. The values in the marker text will dictate which code paths are taken and intrinsically limit the coverage.

Recently, variations on directed fuzz testing have been introduced parallel to the work presented in this paper. Jared DeMott's *Evolutionary Fuzzing System* [10] uses genetic algorithms to construct viable input sets based on reproductive criteria driven by the amount of code coverage of each successive run. It eliminates the risks of wasting effort on unimplemented functionality and of failing to exercise undocumented features. Like fuzz [14], it still must overcome basic protocol input validation tests. Usually, these tests are used in software to determine the format of incoming user input. This might be a version check similar to the protocol banner in OpenSSH [3] or a file format type indicator like the magic check in LibTIFF [2]. While this limitation may not affect the approach dramatically, other techniques, inspired by fuzz testing, address this issue through application flow analysis. Catchconv [15], EXE [8], and SAGE [12] leverage symbolic execution to guide input error detection and generation. Constraints are extracted by tracing the execution of an application on fixed input, such as a known good file. The extracted constraints are then explored through virtualized execution and, in some cases, through repeated execution on input mutated based on code coverage heuristics. These approaches have shown promising results but are limited by approximation errors in symbolic execution and the potential of poor initial input selection.

3.2 Fuzzing flayed applications

Fuzzing flayed applications is a lightweight testing approach which minimizes the initial time investment required from the auditor. The only initial work required is flaying. It does not require a protocol aware input generator, a large testing harness, or any input selection work. Instead, a time investment is required when a crash condition is uncovered. The auditor must spend time creating viable input or determining if the bug is unreachable in normal circumstances.

Flaying is an iterative process for increasing the reachability of complex application code by removing the outer layers of application defenses. Initially, an auditor must supply random input to a target application and analyze the resulting taint tracing output. As uninteresting, or non-state building, sanity and error checks are traversed, they must be forcibly followed or bypassed using Flayer's flow alteration commands. This process is repeated until the desired code is directly exposed for testing. Once exposed, traditional random fuzz testing is used to uncover vulnerabilities. Upon the discovery of a vulnerability, the malicious input must be crafted by the auditor such that it will bypass the removed checks in an unaltered version of the software. The success of this technique is discussed in Section 5.

```
$ valgrind \
     --tool=flayer \
     --taint-network=yes \
     --trace-children=yes \
     --alter-fn=0x8A2E:3 \
     /usr/sbin/sshd -ddd -f \
       $PWD/sshd_config -p 2222 -D
```

Figure 1: Bypassing the "Protocol Mismatch" error check on an Ubuntu Feisty OpenSSH 4.3p2-8ubuntu1 binary

Flayer may be used on an application regardless of the availability of the source code or debugging symbols. While the availability of this data will speed the flaying and creation of valid input, simple heuristics work in many cases which make them unnecessary. For instance, if testing of OpenSSH's cipher suite negotiation is desirable, then it would be useful to bypass the SSH protocol version check. This is done in Figure 1 by stepping over a `sscanf` call. Address `0x8A2E` was identified as the call site to the offending check as it preceded the first tainted call to the logging function which generated the bad protocol version error message. Only the `libc` symbols were used to infer this. With the check removed, it becomes possible to build a simple test harness that copies data from `/dev/urandom` and sends it to the flayed `sshd`. In addition, it is trivial to introduce the required data into any payload by prepending a proper version value. While this is a simplistic example, it captures the essence of the technique.

It is worth noting that the fuzz testing of flayed applications does not require Flayer. This technique was first performed manually through the removal of error and sanity checks using interactive debugging and source code modification. However, the automation of the iterative discovery and modification process greatly speeds the use. The primary benefit of manual flaying is the ability to bypass state building statements through code addition.

4 Further uses

The Flayer tool suite provides a useful feature set for software auditors, developers, and maintainers. The ability to comprehend and interact with the flow of data through an application provides unique insight into that application's operation and makes other useful security auditing and testing techniques possible.

4.1 Guided source code auditing

Many of the more dangerous vulnerabilities, such as remote execution of code, result from malicious user input. Therefore, it is quite useful to determine input entry points and input-tainted functions when auditing an application. This is where Flayer proves useful.

By running a given application, compiled with debugging symbols, through Flayer with an arbitrary input set, the auditor can see which conditional jumps are traversed by the data along with the containing functions. Given that the direct output from Flayer is not always immediately comprehensible to a human auditor, this technique is augmented by the use of FlayerSh.

FlayerSh parses the output of Flayer providing error summaries, branch alteration, and source code snippet listing. Figure 2 provides an example session which shows a run of `tiffinfo` on random input, locations where tainted values were used, and the source code from one such use in a magic value check. Using this shell, it is possible to rapidly follow the data flow as well as review snippets of source code surrounding locations where tainted data was used. This allows for quick insight into the operation of the target application and immediately displays error checking locations without the need for additional tools or software.

FlayerSh does not replace interactive debuggers or disassemblers, such as *GDB* [1] or *IDA Pro* [9], but it does provide a compromise between single stepping through code execution and manually locating application error checking code.

```
$ dd if=/dev/urandom of=rnd.tiff \
  bs=1k count=1
$ FlayerSh ./tiffinfo  /demo/rnd.tiff
>>> filter(file="/demo/rnd.tiff")
>>> run();summary()

==> UninitCondition
id    frame information
0x0 0x4051CC0 TIFFClientOpen
  /demo/libtiff/tif_open.c:359
0x1 0x4051CD0 TIFFClientOpen
  /demo/libtiff/tif_open.c:359
0x2 0x4051CE0 TIFFClientOpen
  /demo/libtiff/tif_open.c:359
0x4 0x413F6A3 _itoa_word
0xd 0x41413B2 vfprintf
0xf 0x413F6BD _itoa_word
==> UninitValue
id  frame information
0x3 0x413F69B _itoa_word
0xe 0x413F6B7 _itoa_word

>>> snippet(0x1, 2)
      * Setup the byte order handling.
      */
|    if (tif->tif_header.tiff_magic !=
           TIFF_BIGENDIAN &&
         tif->tif_header.tiff_magic !=
           TIFF_LITTLEENDIAN
>> alter(0x0, 1)
...
```

Figure 2: A snippet of a guided auditing session in FlayerSh reviewing a magic check in *tiffinfo* (LibTIFF-3.8.2).

4.2 Patch and vulnerability analysis

In complement to auditing and testing, Flayer and FlayerSh, in particular, prove useful when analyzing input data flow through variants of the same piece of software. This scenario occurs quite frequently in both the commercial and open source worlds: projects fork, operating system distributions apply different patches to the same original application, and systems become dependent on old versions of software. When vulnerabilities are announced, patches to the original source code will often not be useful to the maintainers of modified source.

It is possible to run two instances of FlayerSh, one on the patched original application and one on an unpatched variant, with a known bad input. This approach allows one to review the code snippet of each of the conditional jumps along the code path of both versions, and, if needed, to force specific behavior to locate any vulnerable code. Performing this simultaneous analysis results in a quick assessment of the variant's behavior.

Figure 3 provides an example of this. It shows a small piece of a FlayerSh session for a version of LibTIFF patched for the directory offset overflow and one that

```
>>> # LibTIFF 3.8.2 unpatched
>>> snippet(0x2)
   * Read offset to next directory for sequential
   * scans.
   */
   (void) ReadOK(tif, &nextdiroff,
             sizeof (uint32));
 } else {
   toff_t off = tif->tif_diroff;

|if (off + sizeof (uint16) > tif->tif_size) {
     TIFFErrorExt(tif->tif_clientdata, module,
      "%s: Can not read TIFF directory count",
      tif->tif_name);
     return (0);
>>>
```

```
| >>> # LibTIFF 3.8.2 patched
| >>> snippet(0x2)
|
|  /*
|   * Check for integer overflow when
|   * validating the dir_off, otherwise
|   * a very high offset may cause an
|   * OOB read and crash the client.
|   * -- taviso@google.com, 14 Jun 2006.
|   */
| |if (off + sizeof (uint16) > tif->tif_size ||
|     off > (UINT_MAX - sizeof(uint16))) {
|      TIFFErrorExt(tif->tif_clientdata, module,
|       "%s: Can not read TIFF directory count",
|       tif->tif_name);
| >>>
```

Figure 3: Patch analysis of LibTIFF version 3.8.2 using two FlayerSh instances.

is not. In particular, it is displaying the affected tainted conditional where a safety check has been added in one version but is missing in the original.

5 Real world experience

Fuzz testing of flayed applications has been used with some success since the summer of 2006. This work resulted in the discovery of multiple vulnerabilities in well known open source applications:

- Seven vulnerabilities in LibTIFF version 3.8.2 were disclosed [22] [23] [24] [25] [26] [27] [28].
- A remote denial of service vulnerability was discovered [30] in OpenSSH which affected all versions before 4.4.
- An out of band read was discovered [31] in libPNG which affected versions 1.0.6 through 1.2.12.
- A NULL pointer dereference was disclosed [29] in OpenSSL which affected all current clients.

In addition, FlayerSh has been used to determine if variants of LibTIFF and OpenSSH were affected by these vulnerabilities.

5.1 Finding a LibTIFF overflow

One of the recently reported vulnerabilities in LibTIFF resulted from an unchecked integer value which had previously gone unnoticed. The value was that of the TIFF directory entry offset read directly from a supplied TIFF image file. This section provides a simple procedure for finding this vulnerability with Flayer.

The first step is identifying a good test application. For the purposes of this vulnerability, `tiffinfo` is used. LibTIFF version 3.8.2 was downloaded and compiled with debugging symbols. With this completed, the compiled tool is run under Flayer with some random input as seen in Figure 4.

```
$ dd if=/dev/urandom of=test.tiff \
  bs=1k count=1
$ valgrind --tool=flayer \
  --taint-file=yes \
  --file-filter=$PWD/test.tiff \
  ./tiffinfo $PWD/test.tiff
```

Figure 4: Tracing random input through *tiffinfo*

The first run will result in an error message about the TIFF header magic. E.g., "`Not a TIFF or MDI file, ...`". In the Flayer output, there are three tainted conditional jump events which occur prior to the first `printf` call. It is assumed that this call issues the error message. Each of these identified conditional jumps are tested by supplying each instruction pointer address at which the event occur to Flayer. One such test is shown in Figure 5.

```
$ valgrind --tool=flayer \
  --taint-file=yes \
  --file-filter=$PWD/test.tiff \
  --alter-branch=0x4049E66:1 \
  ./tiffinfo $PWD/test.tiff
```

Figure 5: Testing a tainted conditional jump in *tiffinfo*

After some trial and error, it is possible to circumvent the BigTIFF and version error checking resulting in a different error message: "`Can not read TIFF directory count`". With the version checks cleared, the directory count code may be exercised by the test harness provided in Figure 6.

```
#!/bin/bash
while /bin/true; do
 dd if=/dev/urandom \
   of=test.tiff bs=1k \
   count=1
 valgrind --tool=flayer \
  --taint-file=yes \
  --file-filter=$PWD/test.tiff \
  --alter-branch="0x4049E6C:1,
                  0x4049EA6:1" \
  ./tiffinfo ./test.tiff
 if [[ $? -ne 0 && $? -ne 1 ]]
 then; break; fi
done
```

Figure 6: An example Flayer test harness

The test harness is simple but has proved effective with LibTIFF and several other tested applications. However, for this vulnerability, once the directory count error message is triggered, a quick review of the source code at the specified line number reveals an integer overflow. In addition, if the auditor attempted to force the conditional jump with a guard value of `0` at that location, it would have immediately resulted in a segmentation fault.

5.2 The good and the bad

Flayer and flaying have been used extensively for real world application auditing and fuzz testing. With use, the strengths and weaknesses of this tool and related techniques are clear.

For patch analysis and guided auditing, Flayer has worked well for the authors' needs, but auditing style is largely personal preference. With debugging symbols and available source code, however, it has proved a straightforward means for discovering input entry points to an application. This allowed for targeted audits which follow the data flow through the audited application without any initial analysis of the source code. In addition, the ability to step over functions and force conditionals was useful in analyzing foreign binary behavior. It is possible to guide binary analysis by indicating the addresses where interesting behavior occurs and forcing that behavior to continue. In many cases, if the target application crashes, it is possible to infer the data primitives expected by examining the resulting logs.

Fuzzing flayed applications is a highly effective technique for testing binary input such as image files and some network protocols. The values supplied by generating random data from `/dev/urandom` will fully exercise the handlers for the incoming binary code once the blocking checks are removed. However, when the input format is highly structured, such as the ASCII protocol HTTP, this coverage drops off significantly. The likelihood of data originating from `/dev/urandom` generating valid HTTP messages is extremely low. This does not completely discount the use of flaying and Flayer from these scenarios, though. Instead, the fully random data source may be replaced with a somewhat protocol aware payload generator. While a fully protocol aware payload generator may yield the most thorough protocol coverage, merging Flayer with a partially protocol aware generator allows for the execution path taken to be targeted. For example, Flayer may be used to bypass the HTTP version check in order to allow for a HTTP BNF-based fuzzer to generate acceptable data without forcing it to be aware of which versions of the protocol are normally implemented.

Flayer has its own limitations. The largest of these is that skipping sections of code, conditional jump branches or entire functions, may result in missing required runtime state. While this is often not a problem, in some cases values are derived from the source data which need to fall within a small range, and that value is used in subsequent calculations or even memory allocations. When this occurs, Flayer is less useful and manual code modification is required to force correct state. Flayer suffers from another limitation. If a conditional jump is forced, it is forced every time. When that conditional jump determines whether loop should continue, it is possible to lock the application in a never ending loop. Flayer provides no mechanism yet to alter the outcome of a conditional a specific number of times.

A practical limitation of Flayer is that it does not yet provide full coverage of all useful taint source system calls. One notable example is `mmap`. This system call is used to map a file on the file system directly into process memory. Surprisingly, instrumenting this system call has not been necessary in testing and analysis done so far. Given that instrumentation has been added as needed, this is only a minor limitation.

6 Future Work

There are many avenues left to explore with Flayer. Most immediately, Flayer's implementation limitations should be removed. This includes expanding the coverage of tainting input vectors, adding support for conditional jump alteration a controllable number of times, adding network taint filtering, as well as adding an assignment

operator to conditional jumps. In the case of an assignment operator, instead of forcing a jump by replacing the guard value, the actual tainted value would be reassigned to the value it is being tested against. This would address state building challenges in a simple, but effective way.

Other, more challenging, work is possible. One example is the addition of origin tracking of tainted memory. There is a Memcheck code branch which supports this concept, but it does not do so in a way compatible with Flayer. Adding this feature to the existing tool would allow further automated analysis and potentially, the automatic generation of input for interesting code paths. An alternate approach for reaching the same goal would be integrating Flayer's output with a program slicing [33] system. This approach would remove the need for origin tracking while still automatically generating input.

Additional work automating programmatic control flow comprehension is another viable direction. It is possible to automate the process of flaying through brute force flow alteration testing or through the integration with more sophisticated systems. For instance, integration with a code coverage tool would allow for automated runs of Flayer with randomly selected conditional jumps to be optimized. This integration would enable a tree view of the code path and provide pruning of dead end code paths from the analysis enhancing the quality of testing.

Along with these extensions, further integration of Flayer with other fuzz testing techniques will yield very useful results. Flayer may be used to force other fuzz testing software to test more targeted areas of code than they were previously able to. More investigation into the compatibility and benefit will be explored.

7 Conclusions

The Flayer tool suite, built on the Valgrind framework using core concepts from Memcheck, should be added to the toolkit of anyone who regularly performs application auditing or vulnerability patch analysis.

Flayer provides mechanisms to trace input flow through an application and to arbitrarily modify that flow. LibFlayer layers a convenient interface on Flayer. FlayerSh provides a reference tool implemented on LibFlayer. This suite enables multiple security auditing and testing techniques, such as flaying. In concert, these tools and techniques allow one to more effectively audit software.

The Flayer tool suite is a starting point for application auditing and analysis that requires extremely little initial investment while yielding solid results. Even though Flayer is still at an early stage, its techniques have proved their efficacy through the discovery of vulnerabilities in Internet security critical applications, such as OpenSSH and OpenSSL. This software is available for public use and enhancement.

7.1 Availability

This entire tool suite is publicly available licensed under the GPL. It can be downloaded at *http://code.google.com/p/flayer*. Contributions are encouraged.

8 Acknowledgments

Thanks to Google and the Google Security Team for supporting this work and to Chris Evans whose encouragement motivated the creation of this paper. In addition, the authors would like to thank Julian Seward, David Molnar, and Chad Dougherty for kind words and useful guidance.

References

[1] Gnu gdb. http://www.gnu.org/software/gdb/.

[2] Libtiff. http://http://www.remotesensing.org/libtiff/.

[3] Openssh. http://www.openssh.org.

[4] D. Aitel. The advantages of block-based protocol analysis for security testing. http://www.immunitysec.com/resources-papers.shtml, 2002.

[5] D. Aitel. Spike. http://www.immunitysec.com/resources-freesoftware.shtml, 2003.

[6] Beyond Security, Inc. Bestorm 2.0 whitepaper. http://www.beyondsecurity.com/bestorm_whitepaper.html, September 2006.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt. Select: a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM Press.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, Alexandria, Virginia, USA, 2006.

[9] DataRescue sa/nv. Ida pro. http://www.gnu.org/software/gdb/.

[10] J. DeMott and Applied Security, Inc. Evolutionary fuzzing system. http://appliedsec.com/resources.html, May 2007.

[11] M. Eddington. Peach. http://peachfuzz.sourceforge.net/README.txt, May 2004.

[12] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. Technical Report MSR-TR-2007-58, Microsoft, May 2007.

[13] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. Http extensions for distributed authoring – webdav. http://www.ietf.org/rfc/rfc2518.txt, February 1999.

[14] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. pages 32–44, December 1990.

[15] D. A. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, February 4 2007.

[16] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. *sp*, 0:231–245, 2007.

[17] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI 2007*, San Diego, California, USA, June 2007.

[18] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[19] L. "pusscat" Grenier and Lin0xx. Byakugan: Automating exploitation. In *ToorCon Seattle*, Pioneer Square, Seattle, Washington, USA, May 2007.

[20] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen. Protos - systematic approach to eliminate software vulnerabilities. In *Invited presentation at Microsoft Research*, Seattle, USA, May 2002.

[21] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, April 2005.

[22] The MITRE Corporation. CVE-2006-3459. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3459, July 2006.

[23] The MITRE Corporation. CVE-2006-3460. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3460, July 2006.

[24] The MITRE Corporation. CVE-2006-3461. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3461, July 2006.

[25] The MITRE Corporation. CVE-2006-3462. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3462, July 2006.

[26] The MITRE Corporation. CVE-2006-3463. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3463, July 2006.

[27] The MITRE Corporation. CVE-2006-3464. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3464, July 2006.

[28] The MITRE Corporation. CVE-2006-3465. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3465, July 2006.

[29] The MITRE Corporation. CVE-2006-4343. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4343, August 2006.

[30] The MITRE Corporation. CVE-2006-4924. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4924, September 2006.

[31] The MITRE Corporation. CVE-2006-5793. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5793, November 2006.

[32] M. Vuagnoux. Autodafé: an act of software torture. Technical report, Swiss Federal Institute of Technology (EPFL), Cryptograhy and Security Laboratory (LASEC), August 2006.

[33] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, 1979.