In Search Of Shotgun Parsers

Katie Underwood
University of Calgary

Michael Locasto
SRI International

May 25, 2016
Overview

Context

Defining The Shotgun Parser

Tainted Path Length In Android Applications

Our Definition In The Wild

Future Work
WHAT ARE WE LOOKING FOR?
Defining The Shotgun Parser
Why Shotgun?

Input **use** and **recognition** intermixed throughout!
What Are We Looking For?

- Before we go searching for shotgun parsers, we need to know what we’re looking for!
- How will we know a shotgun parser when we see one?
- We frame our definition in the context of static taint analysis of control flow graphs
Hallmarks of the Shotgun Parser

Large Spread Relative To Size
How far does untrusted data propagate through the code?
Hallmarks of the Shotgun Parser

**Large Spread Relative To Size**
How far does untrusted data propagate through the code?

**Use Before Full Recognition**
Is input data fully validated before being used?
Hallmarks of the Shotgun Parser

Large Spread Relative To Size
How far does untrusted data propagate through the code?

Use Before Full Recognition
Is input data fully validated before being used?

Large Number of Variables Involved In Each Tainted Path
How much program state is affected by properties 1 and 2?
Consider an application $A$, which reads a set of untrusted inputs $N$. 

Let $G$ be the static control-flow graph which describes $A$. Let $P_n$ be the connected subgraph induced by the vertices of $G$ tainted by $n \in N$, where $d(P_n) / d(G)$.
Property 1: Spread Relative To Size

- Consider an application $\mathcal{A}$, which reads a set of untrusted inputs $\mathcal{N}$
- Let $G$ be the static control-flow graph which describes $\mathcal{A}$
Property 1: Spread Relative To Size

- Consider an application $A$, which reads a set of untrusted inputs $\mathcal{N}$
- Let $G$ be the static control-flow graph which describes $A$
- Let $P_n$ be the connected subgraph induced by the vertices of $G$ tainted by $n \in \mathcal{N}$, where $d(P_n) \leq d(G)$
Property 1: Spread Relative To Size

- Consider an application $\mathcal{A}$, which reads a set of untrusted inputs $\mathcal{N}$
- Let $G$ be the static control-flow graph which describes $\mathcal{A}$
- Let $P_n$ be the connected subgraph induced by the vertices of $G$ tainted by $n \in \mathcal{N}$, where $d(P_n) \leq d(G)$
- Let $S = \{P_i|1 \leq i \leq |\mathcal{N}|\}$ be the set of all taint-induced subgraphs on $G$
Property 1: Spread Relative To Size

Shotgun parser indicators:

- $d(P_n)$ comparable to $d(G)$
  - Indicates input $n$ not handled in principled manner
Property 1: Spread Relative To Size

Shotgun parser indicators:

- \( d(P_n) \) comparable to \( d(G) \)
  - Indicates input \( n \) not handled in principled manner

- Large \( |S| \)
  - Evidence for presence of multiple shotgun parsers in \( \mathcal{A} \)
Property 2: Use Before Full Recognition

- We can’t quantify whether arbitrary input to an arbitrary piece of code is “fully recognized”

- We *can* start to define a set of standards for handling of specific data types
Property 2: Use Before Full Recognition

For example:

- “For inputs of type O, you must do 5 reads of 4 bytes each, then write 20 bytes in a specific order”
- Identify read/write memory events which take place after input is received
Property 2: Use Before Full Recognition

For example:

- "For inputs of type O, you must do 5 reads of 4 bytes each, then write 20 bytes in a specific order"

- Identify read/write memory events which take place after input is received
Property 3: Number of Tainted Input Variables

- Consider again a tainted subgraph $P_n$
Property 3: Number of Tainted Input Variables

- Consider again a tainted subgraph $P_n$

- Let $P_n$ now be a weighted graph, where each edge $E(x, y)$ corresponds to the number of variables tainted by $n$ after node $x$
Property 3: Number of Tainted Input Variables

Shotgun parser indicators:

- Large number of tainted variables compared to total number of variables
  - Indicates untrusted input affects significant proportion of program state
Property 3: Number of Tainted Input Variables

Shotgun parser indicators:

- Large number of tainted variables compared to total number of variables
  → Indicates untrusted input affects significant proportion of program state

- Areas of $P_n$ where edge weight increases may merit further study
  → Allows us to triage program statements / methods for further analysis
Definition Summary

The "worst case" shotgun parser exhibits all three properties in abundance!
The “worst case” shotgun parser exhibits all three properties in abundance!
CASE STUDY: ANDROID

First Steps Towards Automated Detection
Our Goals

- Establish foundation for a recognizer

- First look at “state of affairs” in Android applications

- Start examining a different class of errors through the LangSec lens
Our Approach

- Static taint analysis of statement-level control flow graphs
- Compute length of tainted path corresponding to each source
- Analysis uses the Jimple intermediate representation

Jimple CFG for one module of the classic game “Snake”
FlowDroid

- Open-source static analysis framework for Android
- Developed by the Secure Software Engineering Group at Paderborn University/ TU Darmstadt

We Add:

- Tracking for *all* tainted paths, not only those terminating in a sink
- Unique identifiers for each taint source
- Specific API call source for each taint
- Taint propagation handler functions to measure input path length

https://blogs.uni-paderborn.de/sse/tools/flowdroid/
Our Implementation

Each time a taint is propagated, our custom handler is invoked:

- Capture incoming flow data object $F$ and outgoing set of flow data objects $\mathcal{F}_{out}$
- If $F$ has not been seen before:
  - Init $F.length = 0$
  - Store original source context of $F$.
- For each flow fact $f \in \mathcal{F}_{out}$:
  - $f.length = F.length + 1$
  - Store source context information for $f$
Workflow

- 56 free apps from 12 different categories

- FlowDroid – Android Data Flow Analysis
  - Modified this open-source tool to calculate tainted path lengths

- Measure lengths of all tainted paths for each application

- Histogram of normalized path lengths
Initial Results
Some Thoughts..

- Our tool is:
  - The foundation of a full SGP recognizer
  - A *prioritization method* for app analysis
OUR DEFINITION IN THE WILD

Let’s Look At Real Stuff
"ImageTragick" (CVE-2016-3714)

```c
sanitize_command=SanitizeDelegateCommand(command);
if (asynchronous == MagickFalse)
    (void) ConcatenateMagickString(sanitize_command, ",&\", MagickPathExtent);
if (message != (char*) NULL)
    *message=\0';
#if defined(MAGICKCORE_POSIX_SUPPORT)
#else defined(MAGICKCORE_HAVE_EXECVP)
    status=system(sanitize_command);
#else
    if ((asynchronous == MagickFalse) ||
        (strpbrk(sanitize_command, ",&\<\>\") != (char*) NULL))
        status=system(sanitize_command);
    else
    {
        pid_t
        child_pid;
        /*
         * Call application directly rather than from a shell.
         */
        child_pid=(pid_t) fork();
        if (child_pid == (pid_t) -1)
            status=system(sanitize_command);
        else
            if (child_pid == 0)
                
                status=execvp(arguments[1], arguments+1);
                _exit(1);
            else
                
                else
```
sanitize_command=SanitizeDelegateCommand(command);
if (asynchronous != MagickFalse)
  (void) ConcatenateMagickString(sanitize_command,"&",MagickPathExtent);
if (message != (char *) NULL)
  *message='\0';
#if defined(MAGICKCORE_POSIX_SUPPORT)
#if !defined(MAGICKCORE_HAVE_EXECVP)
  status=system(sanitize_command);
#else
  if (!(asynchronous != MagickFalse) ||
      (strpbrk(sanitize_command,"&;<>"!)) != (char *) NULL))
    status=system(sanitize_command);
#else
  
  pid_t
    child_pid;

  /*
   * Call application directly rather than from a shell.
   */
  child_pid=(pid_t) fork();
  if (child_pid == (pid_t) -1)
    status=system(sanitize_command);
  else
    if (child_pid == 0)
      { status=execvp(arguments[1],arguments+1);
        _exit(1);
      }
  else

"ImageTragick" (CVE-2016-3714)

```c
sanitize_command=SanitizeDelegateCommand(command);
if (asynchronous != MagickFalse)
    (void) ConcatenateMagickString(sanitize_command "&" MagickPathExtent);

static char *SanitizeDelegateCommand(const char *command)
{
    char
        *sanitize_command;

    const char
        *q;

    register char
        *p;

    static char
        whitelist[] =
            "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_- 
            .@;<>()|/\\"":%=~" 

    sanitize_command=AcquireString(command);
p=sanitize_command;
q=sanitize_command+strlen(sanitize_command);
for (p+=strspn(p,whitelist); p != q; p+=strspn(p,whitelist))
    *p='-' ;
return(sanitize_command);
}
```
"ImageTragick" (CVE-2016-3714)
"ImageTragick" (CVE-2016-3714)
"ImageTragick" (CVE-2016-3714)
"ImageTragick" (CVE-2016-3714)
"ImageTragick" (CVE-2016-3714)

Observations:

- (Relatively) long path
  - 7 direct function calls between input and (attempted) validation, but input is also passed elsewhere
- Raw input is passed between (and used in) 5 different functions before being read into a native data structure
- Input use and validation is intermixed
- Unsuitable validation mechanism
"Heartbleed" (CVE-2014-0160)

tls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                        &s->s3->rrec.data[0], s->s3->rrec.length,
                        s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
    {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 bytes
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);
"Heartbleed" (CVE-2014-0160)

Observations:

- Input passed via several function calls before processing, but not used along the way
- Low degree of input use / validation intermixing, however...
- Almost *total* lack of validation of heartbeat payload!
Mongrel Web Server - HTTP 1.1 Parser

Parsing Done Right!

- Define a finite state machine for HTTP parsing (uses the Ragel compiler)
- Finite state machine $\equiv$ regular grammar
- Input language is correctly, formally defined
- Input data is correctly, formally recognized
In The Context Of Our Definition...

# variables

use before full recognition

tainted path length
In the context of our definition...

- # variables
- Use before full recognition
- Tainted path length
In The Context Of Our Definition...
In The Context Of Our Definition...
FUTURE WORK
Where Do We Go From Here...
Many Roads Lead From Here

- “Climb the hill of Android”
- Develop automated analysis frameworks based on our definition for other software ecosystems
- Develop well-defined input/output patterns for common types (characterize “recognition”)
- Rigorously characterize existing vulnerabilities
- …
We gratefully acknowledge Steven Arzt from the Secure Software Engineering Group at TU Darmstadt for his ongoing assistance with technical questions about FlowDroid via the Soot mailing list.
Other Thoughts...

- Not all vulnerabilities are shotgun parsers...and not all shotgun parsers are necessarily vulnerable

- However:
  - If input data is scattered throughout the code - not just an issue of attack surface, but being error-prone
  - Path length also speaks to how long it takes you to do the parsing - why aren’t you validating as soon as data enters your software?
Practical Issues

- **Platform specific complications**
  - FlowDroid dummy main method - necessary due to Android Lifecycle

- **Abstraction level**
  - Jimple is an intermediate representation

- **Static analysis of real applications is memory intensive!**
  - And we had time constraints...