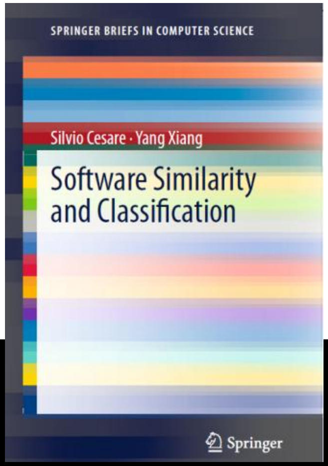
Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis

Silvio Cesare
<silvio.cesare@gmail.com>

Who am I and where did this talk come from?

- Ph.D. Student at Deakin University
- Book Author
- This talk covers some of my Ph.D. research.



Introduction

- Detecting bugs in binary is useful
  - Black-box penetration testing
  - External audits and compliance
  - Verification of compilation and linkage
  - Quality assurance of 3rd party software

Innovation in this work

- Performing static analysis on binaries by:
  - Using decompilation
  - And using data flow analysis on the high level results

- The novelty is in combining decompilation and traditional static analysis techniques

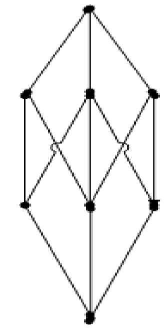
Formal Methods of Program Analysis

- Theorem Proving →

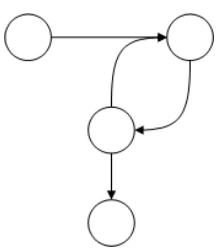
$$\frac{R(r1) \rightarrow n1}{(r3 := r1, P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n1]]} \; ASSIGN$$

$$\frac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

- Abstract Interpretation →



- Model Checking →

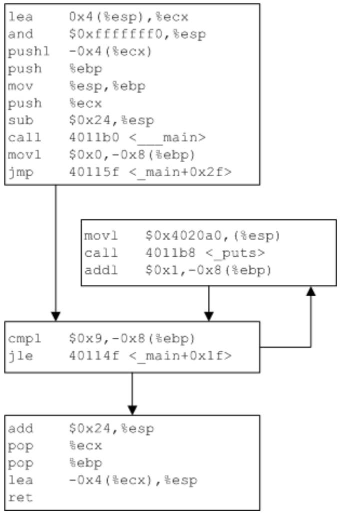


Outline

- Decompilation
- Data Flow Analysis
- IL Optimisation
- Bug Detection
- Bugwise
- Future Work and Conclusion

Terminology (1)

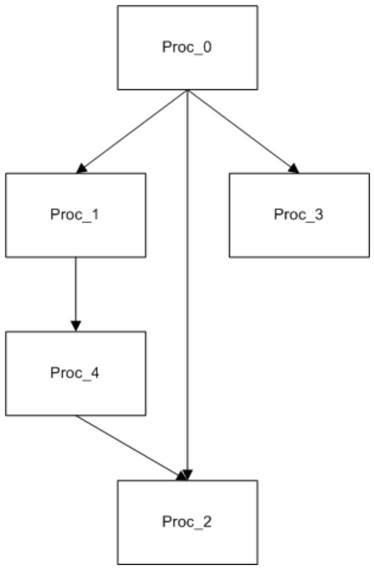
- Control Flow Graphs represents control flow within a procedure
- Intraprocedural analysis works on a single procedure.
  - Flow sensitive analyses take control flow into account
  - Pointer analyses can be flow insensitive



Terminology (2)

- Call Graphs represents control flow between procedures
- Interprocedural analysis looks at all procedures in a module at once
  - Context sensitive analyses take into account call stacks

Proc_0 Proc_1 Proc_4 Proc_2	Proc_0 Proc_2	Proc_0 Proc_3



Decompilation

Decompilation overview

- Recovers source-level information from a binary
- Approach
  - Representing x86 with an intermediate language (IL)
  - Inferring stack pointers
  - Decompiling locals and procedure arguments

Wire – An Formal Language for Binary Analysis

- x86 is complex and big
- Wire is a low level RISC assembly style language
- Translated from x86
- Formally defined operational semantics

$$\frac{R(r1) \rightarrow n1 \quad M(n1) \rightarrow n2}{(r3 := *\,(r1), P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n2]]}\,LOAD$$

The LOAD instruction implements a memory read.

Wire – Equivalence of Dead Code Insertion Obfuscation

Reg_name("eax") = 0
Reg_name("ebx") = 1
Reg_name("zf") = 100

In the first part of the dead code equivalence proof we execute the instructions without the dead code.

$$\frac{n1 = 0}{(\text{"}ASSIGNC\ 0,-,0\text{"}, s) \Longrightarrow s'}$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto n1]\big]$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto 0]\big]$$

In the second part of the proof we execute the instructions with the dead code.

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = n1 + 50 \end{array}}{(\text{"}BOPC_{ADD}\ 0,\$50,0\text{"}, t) \Longrightarrow t'}$$

$$t' = P[pc = pc + 1, R[0 \mapsto n3]]$$

$$t' = P[pc = pc + 1, R[0 \mapsto n1 + 50]]$$

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = n1 - 50 \end{array}}{(\text{"}BOPC\ _{SUB} 0,\$50,0\text{"}, s') \Longrightarrow s''}$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto n3]\big]$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto (n1 + 50) - 50]\big]$$

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = 0 \end{array}}{(\text{"}ASSIGNC\ 0,-,0\text{"}, t'') \Longrightarrow t'''}$$

$$t''' = P\big[pc = pc + 2, R[0 \mapsto n1]\big]$$

$$t''' = P\big[pc = pc + 2, R[0 \mapsto 0]\big]$$

Now we can see that t'''-pc = s'-pc which means they are semantically equivalent when ignoring the effect the code has on the program counter. We also note that s' and s'' are semantically equivalent. We have thus proven the obfuscated and deobfuscate code samples are equivalent.

Stack Pointer Inference

- Proposed in HexRays decompiler - http://www.hexblog.com/?p=42
- Estimate Stack Pointer (SP) in and out of basic block
  - By tracking and estimating SP modifications using linear equalities
- Solve.

// ESP at the entry point is zero

$in_0 = 0$

// ESP at return instructions is zero

$out_4 = 0$

// Equations derived from control flow edges:

$in_1 - out_0 = 0$

$in_2 - out_1 = 0$

$in_3 - out_1 = 0$

$in_3 - out_2 = 0$

$in_4 - out_0 = 0$

// Equations derived from block contents:

$out_0 - in_0 = 0$ // block does not change ESP

$out_1 - in_1 <= 8$ // because of 2 pushes

$out_2 - in_2 = 0$ // block does not change ESP

$out_3 - in_3 = 0$ // block does not change ESP

$out_4 - in_4 = 0$ // block does not change ESP

Picture from HexRays blog.

Local Variable Recovery

- Based on stack pointer inference
- Access to memory offset to the stack
- Replace with native Wire register

```
Imark      ($0x80483f5, , )
AddImm32   (%esp(4), $0x1c, %temp_memreg(12c))
LoadMem32  (%temp_memreg(12c), , %temp_op1d(66))
Imark      ($0x80483f9, , )
StoreMem32(%temp_op1d(66), , %esp(4))
Imark      ($0x80483fc, , )
SubImm32   (%esp(4), $0x4, %esp(4))
LoadImm32  ($0x80483fc, , %temp_op1d(66))
StoreMem32(%temp_op1d(66), , %esp(4))
Lcall      (, , $0x80482f0)
```

```
Imark      ($0x80483f5, , )
Imark      ($0x80483f9, , )
Imark      ($0x80483fc, , )
Free       (%local_28(186bc
```

Procedure Parameter and Argument Recovery

- Based on stack pointer inference
- Offset relative to ESP/EBP indicates local or argument
- Arguments also live registers on procedure entry

```
Free      (%local_28(186bc), , )
Imark     ($0x8048401, , )
Imark     ($0x8048405, , )
Imark     ($0x8048408, , )
PushArg32 ($0x0, %local_28(186bc), )
Args      (, , )
Call      (, , *0x30)
```

Data Flow Analysis

Data Flow Analysis overview

- Data Flow Analysis (DFA) reasons about data
- DFA is conservative
  - It over-approximates
  - But should not under-approximate
- DFA is what an optimising compiler uses
- Analyses
  - Reaching Definitions
  - Upwards Exposed Uses
  - Live Variables
  - Reaching Copies
  - etc

Monotone Frameworks

- Models many data flow problems
- Sets of data entering (in) and leaving (out) of basic blocks
- Set up equations (forwards analysis)
  - Data entering or leaving basic block is initialised
  - Transfer function performs action on data in a basic block

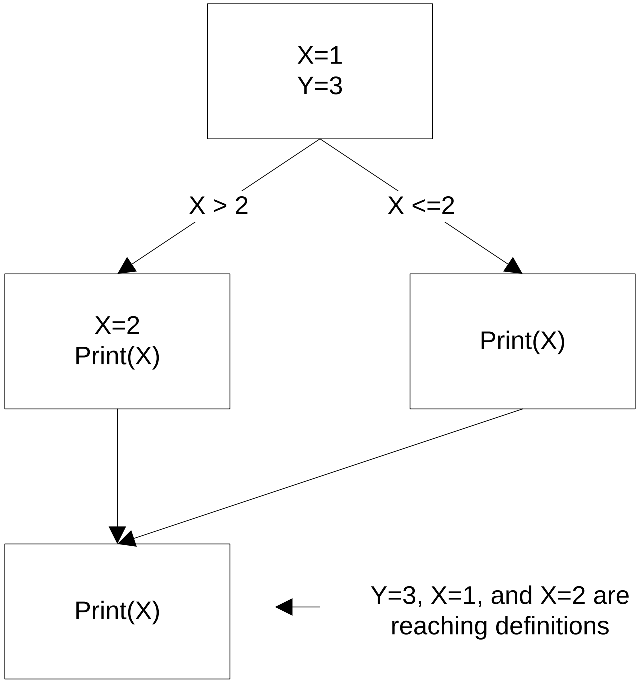
$$out_b = transfer\_function(in_b)$$

  - Join operator combines predecessors in control flow graph

$$in_b = join(\{p \mid p \in predecessor_b\})$$

Reaching Definitions Example

- A reaching definition is a definition of a variable that reaches a program point without being redefined.



A Framework for Data Flow Analysis

- Forwards and backwards analysis
- Initialise in, out, gen, kill sets for each BB.
- Transfer function (forward analysis) is defined as:

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

- Join operator is Union or Intersection.

Reaching Definitions

- Gen and Kill sets
  - gen[B] = { definitions that appear in B and reach the end of B}
  - kill[B] = { all definitions that never reach the end of B}

- Initialisation
  - out[B] = gen[B]

- Confluence Operator
  - Join = Union
  - in[B] = U out[P] for predecessors P of B

Upward Exposed Uses

- The uses of a definition

- Gen and Kill sets
  - gen[B] = { (s,x) | s is a use of x in B and there is no definition of x between the beginning of B and s}
  - kill[B] = { (s,x) | s is a use of x not in B and B contains a definition of x}

- Initialisation
  - in[B] = {0}

- Confluence Operator
  - Join = Union
  - out[B] = U in[S] for successors S of B

More Data Flow Problems

- Live Variables
  - A variable is live if it will be subsequently read without being redefined.
- Reaching Copies
  - The reach of a copy statement
- More DFA analyses used in optimising compilers
  - Available expressions
  - Very busy expressions
  - etc

An Iterative Solution

- Initialise
- Apply transfer function and join.
- Iterate over all nodes in the control flow graph
- Stop when the nodes' data stabilise
- A "Fixed Point"

A Logic-based Solution

- Data flow can be analysed using logic
- Datalog is a syntactic subset of prolog
- Represent analyses and solve

```
Reach(d,x,j):-        Reach(d,x,i),
                      StatementAt(i,s),
                      !Assigns(s,x),
                      Follows(i,j).

Reach(s,x,j):-        StatementAt(i,s),
                      Assigns(s,x),
                      Follows(i,j).
```

Interprocedural Analysis

- Dataflow analysis works on the intraprocedural CFG
- So.. Make an interprocedural CFG (ICFG)
- Replace Calls with branches
- Replace Returns with branches back to callsite
- Apply monotone analysis

IL Optimisation

IL Optimisation overview

- Required to perform other analyses
  - Decompilation
  - Bug Detection

- Reduces the size of IL code

- Optimisations based on data flow analysis
  - Constant Folding and Propagation
  - Copy Propagation
  - Backwards Copy Propagation
  - Dead Code Elimination
  - etc

Constant Folding

- Motivation - replace x=5 + 5 with x=10
- For each arithmetic operator
  - If the reaching definition of each operand is a single constant assignment
  - Fold constants in instruction

Constant Propagation

- Motivation – reduce number of assignments

```
x=34
r=x+y
Print(r)
```

```
r=34+y
Print(r)
```

- If all the reaching definitions of a variable have the same assignment and it is constant:
  - The constant can be propagated to the variable

Copy Propagation

- Motivation – reduce number of copies

```
y=x
z=2
r=y+z
Print(r)
```

```
z=2
r=x+z
Print(r)
```

- For a statement u where x is being used:
  - Statement s is the only definition of x reaching u
  - On every path from s to u there are no assignments to y.
- Or.. At each use of x where x=y is a reaching copy, replace x with y.

Backwards Copy Propagation

- Motivation – reduce number of copies

```
x=34
y=4
r1=x+y
r2=r1
```

```
x=34
y=4
r2=x+y
```

- In Bugwise, both forwards and backwards copy propagation are required.

Dead Code Elimination

- Motivation – reduce number of instructions
- For any definition of a variable:
  - If the variable is not live, then eliminate the instruction.

```
x=34    (x is not live)
x=10
Print(x)
```

```
x=10
Print(x)
```

Bug Detection

Bug detection overview

- Decompilation
  - Transforms locals to native IL variables
- Data Flow Analysis
  - Reasons about IL variables
  - When variables are used and defined
- Bug Detection
  - getenv()
  - Use-after-free
  - Double free

getenv()

- Detect unsafe applications of getenv()
- Example: strcpy(buf,getenv("HOME"))
- For each getenv()
  - If return value is live
  - And it's the reaching definition to the 2nd argument to strcpy()/strcat()
  - Then warn

- P.S. 2001 wants its bugs back.

Use-after-free

- For each free(ptr)
  - If ptr is live
  - Then warn

```
void f(int x)
{
        int *p = malloc(10);
        dowork(p);
        free(p);
        if (x)
                p[0] = 1;
}
```

Double free

- For each free(ptr)
  - If an upward exposed use of ptr's definition is free(ptr)
  - Then warn

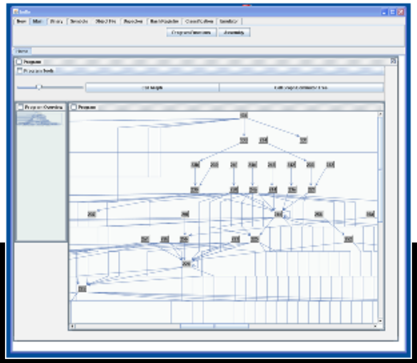
```
void f(int x)
{
        int *p = malloc(10);
        dowork(p);
        free(p);
        if (x)
                free(p);
}
```

- 2001 calls again

Bugwise

Implementation

- Built on my previous Malwise system
- Malwise is over 100,000 LOC C++
- Bugwise is a set of loadable modules
- Everything in this talk and more is implemented



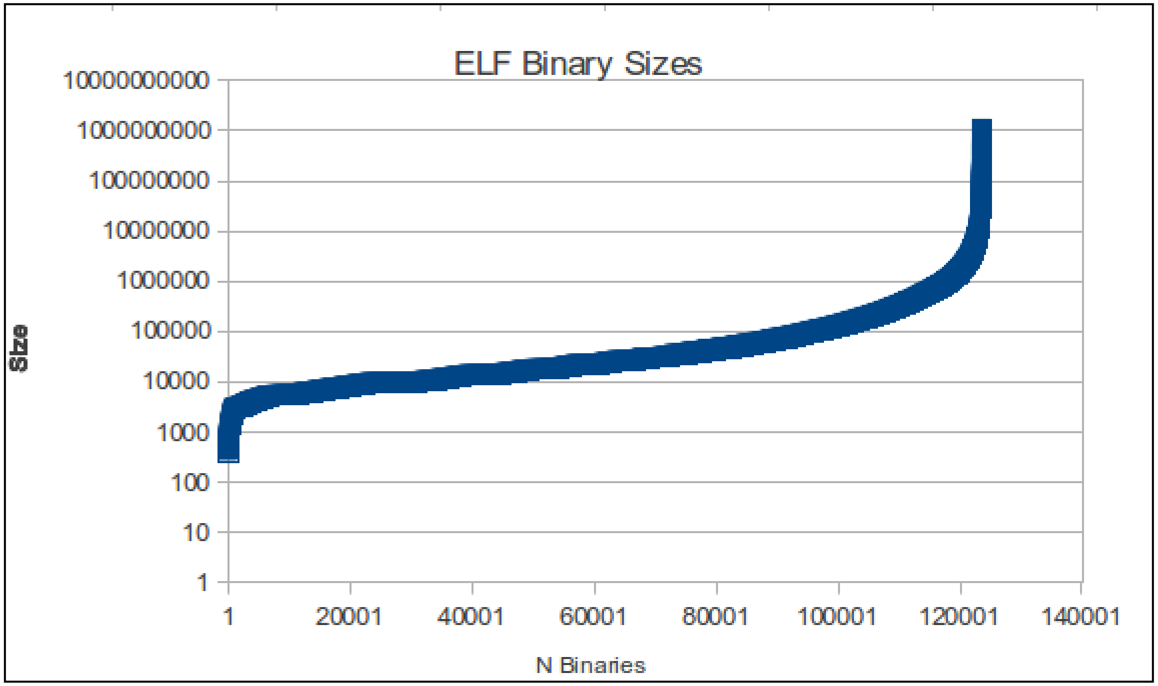
getenv() bugs results

- Scanned entire Debian 7 unstable repository
- ~123,000 ELF binaries
- 30,450 not scanned.
- 85 bug reports
- 47 packages reported

4digits	ptop
acedb-other-belvu	recordmydesktop
acedb-other-dotter	rlplot
bvi	sapphire
comgt	sc
csmash	scm
elvis-tiny	sgrep
fvwm	slurm-llnl-slurmdbd
garmin-ant-downloader	statserial
gcin	stopmotion
gexec	supertransball2
gmorgan	theorur
gopher	twpsk
gsoko	udo
gstm	vnc4server
hime	wily
le-dico-de-rene-cougnenc	wmpinboard
libreoffice-dev	wmppp.app
libxgks-dev	xboing
lie	xemacs21-bin
lpe	xjdic
mp3rename	xmotd
mpich-mpd-bin	
open-cobol	
procmail	

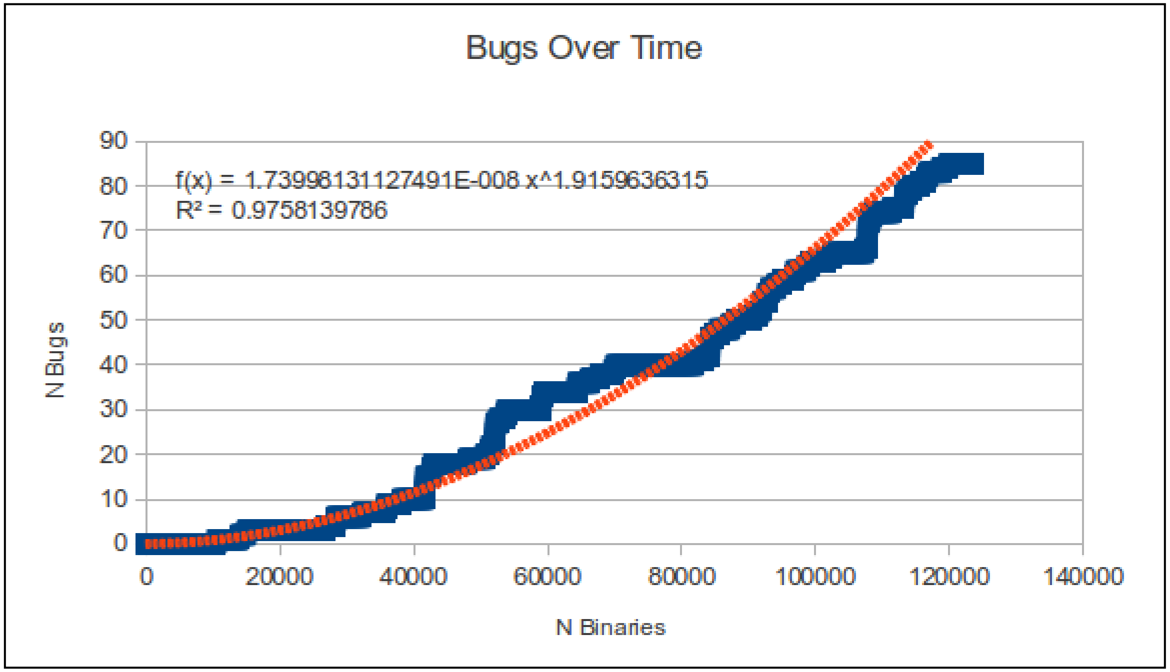
ELF Binary Sizes

- Linear growth with logarithmic scaling plus outliers



Cumulative getenv() bugs over time - sorted by binary size

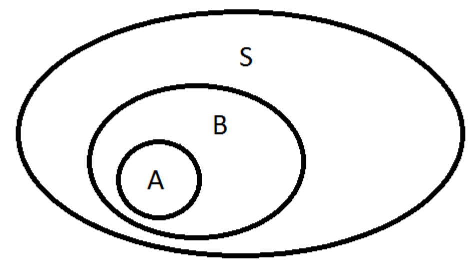
- Linear or power growth?



getenv() bug statistics

- Probability (P) of a binary being vulnerable: 0.00067
- P. of a package being vulnerable: 0.00255

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$



- P. of a package having a 2nd vulnerability given that one binary in the package is vulnerable: 0.52380

Double free SGID games "xonix" in Debian 6

```
memset(score_rec[i].login, 0, 11);
strncpy(score_rec[i].login, pw->pw_name, 10);
memset(score_rec[i].full, 0, 65);
strncpy(score_rec[i].full, fullname, 64);
score_rec[i].tstamp = time(NULL);
free(fullname);

if((high = freopen(PATH_HIGHSCORE, "w",high)) == NULL) {
  fprintf(stderr, "xonix: cannot reopen high score file\n");
  free(fullname);
  gameover_pending = 0;
  return;
}
```

Bugalyze.com

www.bugalyze.com/webservices/Bugwise/Bugwise-print-report?h=f6dcd3c6cdda0820910256afb78c2449

Bugwise A binary-level bug detection service by FooCodeChu.

Submission Details

Hash	f6dcd3c6cdda0820910256afb78c2449

Results Summary (Permanent link to this report)

1 Double frees detected.

Results

Double free	Double free at 0x804c910 and 0x804cc8c

www.bugalyze.com/webservices/Bugwise/Bugwise-print-report?h=9e8a7c678b3495cf0eeef02ce691d47a

Bugwise A binary-level bug detection service by FooCodeChu.

Submission Details

Hash	9e8a7c678b3495cf0eeef02ce691d47a

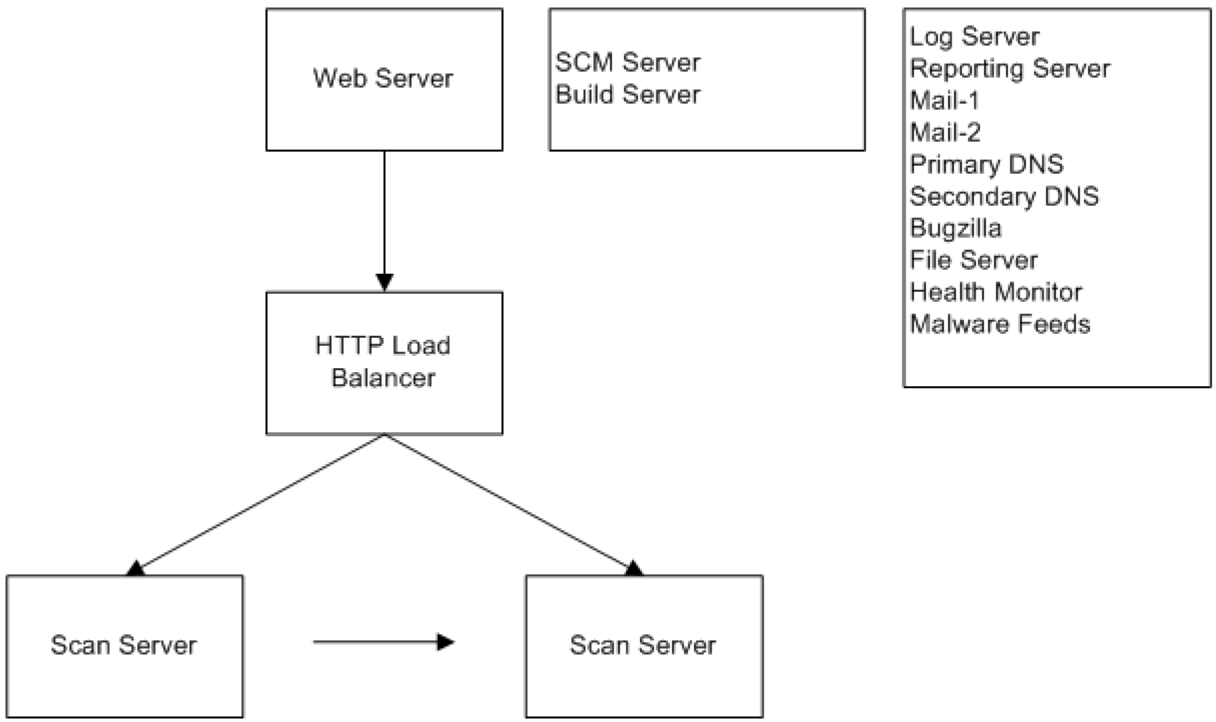
Results Summary (Permanent link to this report)

1 Buffer overflows detected.

Results

Buffer overflow	Results of getenv() overflow strcpy function

EC2 Infrastructure



Future work and conclusion

Future Work

- Core
  - Summary-based interprocedural analysis
  - Context sensitive interprocedural analysis
  - Pointer analysis
  - Improved decompilation

- Bug Detection
  - Uninitialised variables
  - Unchecked return values
  - More evaluation and results

Conclusion

- Traditional static analysis can find bugs.
- Decompilation bridges the binary gap.
- Bugwise works on real Linux binaries.
- It is available to use.
- http://www.Bugalyze.com