The Parfait Bug-Checker

Cristina Cifuentes, Nathan Keynes, Lian Li

Sun Microsystems Laboratories
Brisbane, Australia

2 October, 2009
Bugs are Part of Life

Vulnerability Note VU#388289

Sun Microsystems Java GIF image processing buffer overflow

Overview
A vulnerability in the Sun Java Runtime Environment may allow an attacker to execute arbitrary code on a

I. Description
The Sun Java Runtime Environment (JRE) allows users to run Java applications in a browser or as stand-alone.
GIF image with a specified width of 0 is processed, the Sun JRE will overwrite memory contents, which can
Note that exploit code for this vulnerability is publicly available.

II. Impact
A remote unauthenticated attacker may be able to execute arbitrary code.

III. Solution

Apply an update
Per Sunsolve document 107270, this issue is addressed in
- JRE and JRE 5.0 Update 10 or later
- SDK and JRE 1.4.2_13 or later
- SDK and JRE 1.3.1_19 or later

Disable Java
Disable Java in your web browser, as specified in the Security Guide Web Browser document.

Systems Affected
Vendor | Status | Date Notified | Date Updated

National Cyber Alert System

Technical Cyber Security Alert TA07-059A

Sun Solaris Telnet Worm

Original release date: February 28, 2007
Last revised: Source: US-CERT

Systems Affected
- Sun Solaris 10 (SunOS 5.10)
- Sun “Nevada” (SunOS 5.11)
Both SPARC and Intel (x86) architectures are affected.

Overview
A worm is exploiting a vulnerability (VU881872) in the Sun Solaris telnet daemon (inetd). Sun has published information about the worm in the Security Alert Feed including an inoculation script that disables the telnet daemon and reverses known changes made by the worm.

© 2009, Sun Microsystems, Inc.
Various Bug-Checking Tools Available in the Market

PolySpace
Digital WRL
ESC
Microsoft Research Slam
VERACODE
GRAMMATech
Codesonar
Fortify
coverity
Prefast
Microsoft
Klocwork

clang
blast
jlint
jpf
splint
uno
findbugs

Commercial
Open Source

© 2009, Sun Microsystems, Inc.
Why Aren’t These Tools Used at Sun?

- Long-running times over MLOC
  - Up to 1 week over ~6 MLOC
- Large false positive rate in practice
  - 30-50%
- High cost for commercial tools given the above
  - Proportional to # LOC
  - Tied to specific software to be checked
  - Maintenance fee on a yearly basis
Sample Source Code Sizes at Sun

• Vast majority is C/C++ system code
  ‣ ...
  ‣ **JDK™** platform
    - 900 KLOC (VM and native libs)
  ‣ ...
  ‣ **OpenSolaris™** operating system
    - OS/Networking (ON) consolidation: 10 MLOC
    - Full distro: >20 MLOC
The Parfait Design
Key Features of the Parfait Design

• Scalability achieved by
  ‣ Layered approach
  ‣ Demand-driven analyses
  ‣ Multiple ways to parallelize framework
    - per bug-type basis, per analysis, per “executable”-file basis

• Precision achieved by
  ‣ Multiple lists of bugs
  ‣ Bugs moved from PotentialBugs to RealBugs list conservatively
Layered analyses:
- ordered, cheap to expensive
- sound analyses
- demand-driven
Layers of Analysis by Example

Finding buffer overflow
3 layers
Sample Program

```c
#include <stdlib.h>
#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    if (argc != 3) {
        printf("Usage: name length data\n");
        exit(-1);
    }

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }

    for (i = 0; i <= n; i++) {
        buf2[i] = argv[2][i];
    }

    return 0;
}
```
```c
#include <stdlib.h>
#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    if (argc != 3){
        printf("Usage: name length data\n");
        exit(-1);
    }

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }

    return 0;
}
```
```c
#include <stdlib.h>

#define BUFF_SIZE 100

int main (int argc, char *argv[])
{
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }

    buf[n] = '\0';
}```
```c
#include <stdlib.h>

#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
        buf[n] = '\0';
    }
}
```

Layer 1 - Constant Propagation & Index Checks
```c
#include <stdlib.h>
#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
  char buf[BUFF_SIZE], *buf2;
  int n = BUFF_SIZE, i;

  for (i = 1; i <= n; i++) {
    buf[i] = 'A';
  }
  buf[n] = '\0';
}
```
#include <stdlib.h>

#define BUFF_SIZE 100

int main (int argc, char *argv[])
{
    int n = BUFF_SIZE, i;

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }
    buf[n] = '\0';

    for (i = 1; i <= 100; i++) {
        if (i < 0 || i > 99)
            return (true);
    }
    return (false);
}
```c
#include <stdlib.h>
#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    n = atoi(argv[1]);
    buf2 = (char*)malloc(n);
    for (i = 0; i <= n; i++) {
        buf2[i] = argv[2][i];
    }
    return 0;
}
```
```c
#include <stdlib.h>

#define BUFF_SIZE 100

int main (int argc, char *argv[]) { char buf[BUFF_SIZE], *buf2;
  int n = BUFF_SIZE, i;
  n = atoi(argv[1]);
  buf2 = (char*)malloc(n);
  for (i = 0; i <= n; i++) {
    buf2[i] = argv[2][i];
  }
  return 0;
}
```

Layer 3 - Symbolic Analysis

Sample Program

© 2009, Sun Microsystems, Inc.
```c
#include <stdlib.h>

#define BUFF_SIZE 100

int main (int argc, char *argv[]) {
    char buf[BUFF_SIZE], *buf2;
    int n = BUFF_SIZE, i;

    if (argc != 3){
        printf("Usage: name length data\n");
        exit(-1);
    }

    for (i = 1; i <= n; i++) {
        buf[i] = 'A';
    }

    for (i = 0; i <= n; i++) {
        buf2[i] = argv[2][i];
    }

    buf[n] = '\0';

    buf2 = (char*)malloc(n);

    for (i = 0; i <= n; i++) {
        buf2[i] = argv[2][i];
    }

    return 0;
}
```
The Parfait Implementation
The Parfait Implementation

- Built on LLVM
- parfait-gcc is a script around llvm-gcc
- parfait is the bug-checker proper

© 2009, Sun Microsystems, Inc.
Pass Dependencies in Parfait

Buffer Overflow List
Augmented SSA
Compiletime Computable
Gated SSA
Symbolic Value
Symbolic Symbol Table
Dataflow Framework
Memory Leak Checks
Pointer Bug List

Constant Index Checks
Partial Evaluation
Symboleic Analysis

Execution Engine
Interpreter
JIT

Dominator Tree
Post Dominance Frontier
Post Dominator Tree
Loop Info
Loop Pass

Promote Memory To Register
Constant Propagation Pass
CFG Simplification Pass
Scalar Evolution

Parfait
LLVM
Results

Reference dataset: OpenSolaris™ ON b93
Latest dataset: Solaris ON b121
Parfait 0.2.3.584 (18 Sep 09)
**Performance Results over Solaris ON b121**

**AMD Opteron 2.8 GHz, 2 GB memory**

<table>
<thead>
<tr>
<th>Build</th>
<th>Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal build</td>
<td>364</td>
</tr>
<tr>
<td>(Sun™ Studio, gcc)</td>
<td></td>
</tr>
<tr>
<td>Parfait build</td>
<td>534</td>
</tr>
<tr>
<td>(Sun Studio, gcc, parfait-gcc)</td>
<td></td>
</tr>
<tr>
<td>Parfait analysis</td>
<td>21</td>
</tr>
<tr>
<td>(parfait)</td>
<td></td>
</tr>
</tbody>
</table>

* 10 million non-commented lines of C/C++ source code (uts, cmd, lib, common, closed)
# Accuracy Results over OpenSolaris b93*

## True Positives and False Positives

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Parfait reports</th>
<th>TP (%)</th>
<th>FP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer overrun</td>
<td>488</td>
<td>93%</td>
<td>7%</td>
</tr>
<tr>
<td>Memory leak</td>
<td>464</td>
<td>92%</td>
<td>8%</td>
</tr>
<tr>
<td>Format string type mismatch</td>
<td>1,009</td>
<td>96%</td>
<td>4%</td>
</tr>
</tbody>
</table>

* 7 million non-commented lines of C/C++ source code (uts, cmd, lib, common, closed)
# Results with Open Source Kernels

**OpenSolaris, Linux, OpenBSD**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (min)</th>
<th>Part</th>
<th>LOC</th>
<th>Buffer overflow</th>
<th>Bug density</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSolaris UTS b105</td>
<td>5</td>
<td>Core</td>
<td>2.1M</td>
<td>15</td>
<td>0.0069</td>
<td>Being fixed</td>
</tr>
<tr>
<td>Device drivers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux 2.6.29*</td>
<td>13</td>
<td>Core</td>
<td>1.6M</td>
<td>12</td>
<td>0.0073</td>
<td>Fixed</td>
</tr>
<tr>
<td>Device drivers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenBSD 4.4</td>
<td>2</td>
<td>Core</td>
<td>0.5M</td>
<td>3</td>
<td>0.0060</td>
<td>Fixed</td>
</tr>
<tr>
<td>Device drivers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Linux has the benefit of 2 separate scans already made by Coverity over the Linux code base
The Parfait User Interface
Web-based GUI
Web-based GUI

• GUI tested with
  ‣ Firefox 3, 3.5
  ‣ Safari 4.0
  ‣ Chrome 1, 2, 3
  ‣ Internet Explorer 8, 7

• GUI tested on
  ‣ Solaris, Mac OS X, Linux and Windows

• Usability testing conducted
  ‣ University students
  ‣ Sun engineers
LLVM Evaluation
Benefits

• Modern compilation infrastructure
  ‣ modular
  ‣ uses SSA representation
  ‣ extensible
• Cross platform
• Portable IR
• Well documented
• Ease of prototyping
Challenges

- Lack of union type information in IR
- Lack of backwards compatibility
- Limited support for debug information
- Memory consumption issues
- Performance issues with llvm-ld
- Reliance on “newer” versions of gcc (4.x)
  - much legacy code doesn’t compile with gcc 4.x
- Some non-extensible implementations
Challenges

llvm-ld overhead is not linear
Summary
Summary

• Parfait is a new C/C++ bug-checking tool
  ‣ scalable and precise
  ‣ starting to be widely used internally
  ‣ external requests

• Extensible framework
  ‣ Our emphasis
    - Buffer overflow, pointer/memory-related errors, format string
  ‣ Our collaborators
    - Concurrency bugs, automated testing, OO-specific bugs

• Has found real bugs in
  ‣ Solaris, OpenBSD, Linux, JDK, ...
The Parfait Team
parfait-dev@sun.com

http://research.sun.com/projects/parfait