CS 6V81.005
Automatic Exploit Generation (AEG)

Matthew Stephen

Department of Computer Science
University of Texas at Dallas

February 20th, 2012
Outline

1. Overview
   - Introduction
   - Considerations

2. AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3. Conclusion
1. Overview
   - Introduction
   - Considerations

2. AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3. Conclusion
Introduction

Current Method
- Think hard about whether a bug is exploitable
- Attempt to create an exploit
## Introduction

### Current Method
- Think hard about whether a bug is exploitable
- Attempt to create an exploit

End-to-end systems not shown to be practical before
Introduction

Current Method
- Think hard about whether a bug is exploitable
- Attempt to create an exploit

End-to-end systems not shown to be practical before

The AEG Method
- Reproduced known exploits
- Found zero-day vulnerabilities

Implications
- Untrained attackers can succeed
- Easy to find zero-day vulnerabilities
- Can also use for defense
Introduction

Current Method
- Think hard about whether a bug is exploitable
- Attempt to create an exploit

End-to-end systems not shown to be practical before

The AEG Method
- Reproduced known exploits
- Found zero-day vulnerabilities

Implications
- Untrained attackers can succeed
- Easy to find zero-day vulnerabilities
- Can also use for defense
1 Overview
   - Introduction
   - Considerations

2 AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3 Conclusion
Considerations

Example code

```c
char src[12], dst[10];
strncpy(src, dst, sizeof(src));
```
Example code

- char src[12], dst[10];
- strncpy(src, dst, sizeof(src));

Can you create a buffer overflow?
Considerations

Example code

- `char src[12], dst[10];`
- `strncpy(src, dst, sizeof(src));`

Can you create a buffer overflow?

If the compiler page-aligns both buffers to 16 bytes, the dst buffer will be large enough for a 12 byte src buffer.
What is Needed?

- Source code
- Executable binary
- LLVM bytecode
What is Needed?

- Source code
- Executable binary
- LLVM bytecode

- From the previous slide, we can see that source code analysis alone is inadequate
What is Needed?

- Source code
- Executable binary
- LLVM bytecode

From the previous slide, we can see that source code analysis alone is inadequate

Binary analysis alone is unscalable
What is Needed?

- Source code
- Executable binary
- LLVM bytecode

From the previous slide, we can see that source code analysis alone is inadequate
- Binary analysis alone is unscalable
- Bytecode needed for bug-finding infrastructure
Outline

1. Overview
   - Introduction
   - Considerations

2. AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3. Conclusion
Outline

1. Overview
   - Introduction
   - Considerations

2. AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3. Conclusion
Challenges

- State space explosion problem
  - Loops greatly increase the number of execution paths (potentially infinite)
Challenges

- State space explosion problem
  - Loops greatly increase the number of execution paths (potentially infinite)
- Path selection problem
  - Which paths to try first?
Challenges

- State space explosion problem
  - Loops greatly increase the number of execution paths (potentially infinite)
- Path selection problem
  - Which paths to try first?
- Environment modeling problem
  - Model environment IO behavior (arguments, files, packets)
Challenges

- State space explosion problem
  - Loops greatly increase the number of execution paths (potentially infinite)
- Path selection problem
  - Which paths to try first?
- Environment modeling problem
  - Model environment IO behavior (arguments, files, packets)
- Mixed analysis challenge
  - Combining analysis of source code and binary
Challenges

- State space explosion problem
  - Loops greatly increase the number of execution paths (potentially infinite)
- Path selection problem
  - Which paths to try first?
- Environment modeling problem
  - Model environment IO behavior (arguments, files, packets)
- Mixed analysis challenge
  - Combining analysis of source code and binary
- Exploit verification problem
  - Ensure that the generated exploit actually works
iwconfig Example

```c
int main(int argc, char **argv) {
    int skfd;         /* generic raw socket desc. */
    if(argc == 2) {
        print_info(skfd, argv[1], NULL, 0);
    ... 
    static int print_info(int skfd, char *ifname, char *args[], int count) {
        struct wireless_info info;
        int rc;
        rc = get_info(skfd, ifname, &info);
    ... 
    static int get_info(int skfd, char *ifname, struct wireless_info *info) {
        struct iwreq wrq;
        if(iw_get_ext(skfd, ifname, SIOCGPIO, &wrq) < 0) {
            struct ifreq ifr;
            strcpy(ifr.ifr_name, ifname); /* buffer overflow */
    ... 
```

- Search for bugs at source code level
iwconfig Example

```c
1 int main(int argc, char **argv) {
2     int skfd; /* generic raw socket desc. */
3     if(argc == 2)
4         print_info(skfd, argv[1], NULL, 0);
5     ...
6 static int print_info(int skfd, char *ifname, char *args[], int count)
7     {
8         struct wireless_info info;
9         int rc;
10        rc = get_info(skfd, ifname, &info);
11    ...
12 static int get_info(int skfd, char *ifname, struct wireless_info *info)
13     {
14         struct iwreq wrq;
15         if(iw_get_ext(skfd, ifname, SIOCGIFNAME, &wrq) < 0) {
16             struct ifreq ifr;
17             strcpy(ifr.ifr_name, ifname); /* buffer overflow */
18             ...
```

- Search for bugs at source code level
- Perform dynamic binary analysis (extract address of buffer and return)
iwconfig Example

1 int main(int argc, char **argv) {
2  int skfd;   /* generic raw socket desc. */
3  if(argc == 2)
4     print_info(skfd, argv[1], NULL, 0);
5  ...
6 static int print_info(int skfd, char *ifname, char *args[], int count)
7  {
8   struct wireless_info info;
9   int    rc;
10  rc = get_info(skfd, ifname, &info);
11 ...
12 static int get_info(int skfd, char *ifname, struct wireless_info *info
13  ) {
14   struct iwreq    wrq;
15   if(iw_get_ext(skfd, ifname, SIOCGIFNAME, &wrq) < 0) {
16    struct ifreq  ifr;
17    strcpy(ifr.ifr_name, ifname); /* buffer overflow */
18  ...

- Search for bugs at source code level
- Perform dynamic binary analysis (extract address of buffer and return)
- Generate constraints describing exploit
iwconfig Example

```c
int main(int argc, char **argv) {
    int skfd; /* generic raw socket desc. */
    if (argc == 2)
        print_info(skfd, argv[1], NULL, 0);
...
static int print_info(int skfd, char *ifname, char *args[], int count)
{
    struct wireless_info info;
    int rc;
    rc = get_info(skfd, ifname, &info);
    ...
static int get_info(int skfd, char *ifname, struct wireless_info *info)
{
    struct iwreq wrq;
    if (iw_get_ext(skfd, ifname, SIOCGIFNAME, &wrq) < 0) {
        struct ifreq ifr;
        strcpy(ifr.ifr_name, ifname); /* buffer overflow */
        ...
```

- Search for bugs at source code level
- Perform dynamic binary analysis (extract address of buffer and return)
- Generate constraints describing exploit
- Queries a constraint solver for satisfying answer
iwconfig Example

```c
1 int main(int argc, char **argv) {
2     int skfd; /* generic raw socket desc. */
3     if(argc == 2)
4         print_info(skfd, argv[1], NULL, 0);
5 ...
6 static int print_info(int skfd, char *ifname, char *args[], int count)
7         {
8             struct wireless_info info;
9             int rc;
10            rc = get_info(skfd, ifname, &info);
11 ...
12 static int get_info(int skfd, char *ifname, struct wireless_info *info
13         ) {
14             struct iwreq wrq;
15             if(iw_get_ext(skfd, ifname, SIOCGIFNAME, &wrq) < 0) {
16                 struct ifreq ifr;
17                 strcpy(ifr.ifr_name, ifname); /* buffer overflow */
18 ...
```

- Search for bugs at source code level
- Perform dynamic binary analysis (extract address of buffer and return)
- Generate constraints describing exploit
- Queries a constraint solver for satisfying answer
- You’re root!
Outline

1 Overview
   • Introduction
   • Considerations

2 AEG
   • Challenges
   • iwconfig Example
   • Constraint Equation
   • Components
   • Preconditioned Symbolic Execution

3 Conclusion
The equation

\[ \pi_{\text{bug}}(x) \land \pi_{\text{exploit}}(x) \Rightarrow \pi_{\text{prec}}(x) \]

Predicates

- \( \pi_{\text{bug}} \): unsafe path predicate - path of an execution that violates safety property
  - Out of bounds writes
  - Unsafe format strings

- \( \pi_{\text{exploit}} \): attacker logic - where to go after hijacking eip
  - Crash the program
  - Perform control flow hijack

- \( \pi_{\text{prec}} \): preconditions - length, prefix, etc.
## Components

### Six components

- **Pre-Process:** $src \rightarrow (B_{gcc}, B_{llvm})$
Components

Six components

- Pre-Process: $src \rightarrow (B_{gcc}, B_{llvm})$
- Src-Analysis: $B_{llvm} \rightarrow max$
  - max is about 10% larger than the largest statically allocated buffer
Components

Six components

- **Pre-Process**: $src \rightarrow (B_{gcc}, B_{llvm})$
- **Src-Analysis**: $B_{llvm} \rightarrow max$
  - max is about 10% larger than the largest statically allocated buffer
- **Bug-Find**: $(B_{llvm}, \phi, max) \rightarrow (\pi_{bug}, V)$
  - $\phi$ is the safety properties
  - $V$ is the source code level information about the vulnerable function

DBA: $(B_{gcc}, (\pi_{bug}, V)) \rightarrow R$
- $R$ is vulnerable buffer address and address of the vulnerable function return address

Exploit-Gen: $(\pi_{bug}, R) \rightarrow \pi_{bug} \land \pi_{exploit}$

Verify: $(B_{gcc}, \pi_{bug} \land \pi_{exploit}) \rightarrow \{\epsilon, \bot\}$
- Returns either a working exploit or nothing
Components

Six components

- **Pre-Process**: \( src \rightarrow (B_{gcc}, B_{llvm}) \)
- **Src-Analysis**: \( B_{llvm} \rightarrow max \)
  - \( max \) is about 10% larger than the largest statically allocated buffer
- **Bug-Find**: \( (B_{llvm}, \phi, max) \rightarrow (\pi_{bug}, V) \)
  - \( \phi \) is the safety properties
  - \( V \) is the source code level information about the vulnerable function
- **DBA**: \( (B_{gcc}, (\pi_{bug}, V)) \rightarrow R \)
  - \( R \) is vulnerable buffer address and address of the vulnerable function return address
Components

Six components

- **Pre-Process**: $src \rightarrow (B_{gcc}, B_{llvm})$
- **Src-Analysis**: $B_{llvm} \rightarrow max$
  - $max$ is about 10% larger than the largest statically allocated buffer
- **Bug-Find**: $(B_{llvm}, \phi, max) \rightarrow (\pi_{bug}, V)$
  - $\phi$ is the safety properties
  - $V$ is the source code level information about the vulnerable function
- **DBA**: $(B_{gcc}, (\pi_{bug}, V)) \rightarrow R$
  - $R$ is vulnerable buffer address and address of the vulnerable function return address
- **Exploit-Gen**: $(\pi_{bug}, R) \rightarrow \pi_{bug} \land \pi_{exploit}$
Components

Six components

- **Pre-Process:** $src \rightarrow (B_{gcc}, B_{llvm})$
- **Src-Analysis:** $B_{llvm} \rightarrow max$
  - max is about 10% larger than the largest statically allocated buffer
- **Bug-Find:** $(B_{llvm}, \phi, max) \rightarrow (\pi_{bug}, V)$
  - $\phi$ is the safety properties
  - $V$ is the source code level information about the vulnerable function
- **DBA:** $(B_{gcc}, (\pi_{bug}, V)) \rightarrow R$
  - $R$ is vulnerable buffer address and address of the vulnerable function return address
- **Exploit-Gen:** $(\pi_{bug}, R) \rightarrow \pi_{bug} \wedge \pi_{exploit}$
- **Verify:** $(B_{gcc}, \pi_{bug} \wedge \pi_{exploit}) \rightarrow \{\epsilon, \bot\}$
  - Returns either a working exploit or nothing
Preconditioned Symbolic Execution

- Manage state space explosion by restricting the paths that are explored
- Assumes input[i] != null-character for 0 < i < 40 and input[41] = null-character

```c
1 int process_input(char input[42])
2 char buf[20];
3 while (input[i] != '\0')
4     buf[i++] = input[i];
```
Types of preconditions

- None
Types of preconditions

- None
- Length
Types of preconditions

- None
- Length
- Prefix
Types of preconditions

- None
- Length
- Prefix
- Concolic execution
  - The path that follows a specific concrete input is taken
Types of preconditions

- None
- Length
- Prefix
- Concolic execution
  - The path that follows a specific concrete input is taken

![Example](image)

(a) An example that illustrates the advantages of preconditioned symbolic execution.

<table>
<thead>
<tr>
<th>Precondition</th>
<th>Input Space</th>
<th># of Interpreters</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>$256^S$</td>
<td>$2^N \cdot S \cdot 2^M$</td>
</tr>
<tr>
<td>Known Length</td>
<td>$255^S$</td>
<td>$2^N \cdot 2^M$</td>
</tr>
<tr>
<td>Known Prefix</td>
<td>$256^S - P$</td>
<td>$2^N - P \cdot (S - P) \cdot 2^M$</td>
</tr>
<tr>
<td>Concolic</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) The size of the input space and the number of interpreters required to explore the state space of the example program at the left, for each of the 4 preconditions supported by AEG. We use $S$ to denote the number of symbolic input bytes and $P$ for the length of the known prefix ($P < N < S$).
Path Prioritization

- Buggy path first
  - Assumes exploitable bugs follow nonexploitable ones
Path Prioritization

- Buggy path first
  - Assumes exploitable bugs follow nonexploitable ones
- Loop exhaustion
  - Higher priority to maximum number of iterations
Dynamic Binary Analysis

- **Stack restoration**
  - Ensure that the stack is not corrupted (results in a crash)
- **Return-to-stack and return-to-libc attacks**
- **Calculate difference in size of stack during analysis and a normal run**
## Effectiveness

<table>
<thead>
<tr>
<th>Program</th>
<th>Ver.</th>
<th>Exploit Type</th>
<th>Vulnerable Input src</th>
<th>Gen. Time (sec.)</th>
<th>Executable Lines of Code</th>
<th>Advisory ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>aeon</td>
<td>0.2a</td>
<td>Local Stack</td>
<td>Env. Var</td>
<td>3.8</td>
<td>3392</td>
<td>CVE-2005-1019</td>
</tr>
<tr>
<td>iwconfig</td>
<td>V.26</td>
<td>Local Stack</td>
<td>Arguments</td>
<td>1.5</td>
<td>11314</td>
<td>CVE-2003-0947</td>
</tr>
<tr>
<td>glftpd</td>
<td>1.24</td>
<td>Local Stack</td>
<td>Arguments</td>
<td>2.3</td>
<td>6893</td>
<td>OSVDB-ID#16373</td>
</tr>
<tr>
<td>ncompress</td>
<td>4.2.4</td>
<td>Local Stack</td>
<td>Arguments</td>
<td>12.3</td>
<td>3198</td>
<td>CVE-2001-1413</td>
</tr>
<tr>
<td>htget (processURL)</td>
<td>0.93</td>
<td>Local Stack</td>
<td>Arguments</td>
<td>57.2</td>
<td>3832</td>
<td>CVE-2004-0852</td>
</tr>
<tr>
<td>htget (HOME)</td>
<td>0.93</td>
<td>Local Stack</td>
<td>Env. Var</td>
<td>1.2</td>
<td>3832</td>
<td>Zero-day</td>
</tr>
<tr>
<td>expect (DOTDIR)</td>
<td>5.43</td>
<td>Local Stack</td>
<td>Env. Var</td>
<td>187.6</td>
<td>458404</td>
<td>Zero-day</td>
</tr>
<tr>
<td>expect (HOME)</td>
<td>5.43</td>
<td>Local Stack</td>
<td>Env. Var</td>
<td>186.7</td>
<td>458404</td>
<td>OSVDB-ID#60979</td>
</tr>
<tr>
<td>socat</td>
<td>1.4</td>
<td>Local Format</td>
<td>Arguments</td>
<td>3.2</td>
<td>35799</td>
<td>CVE-2004-1484</td>
</tr>
<tr>
<td>tipxd</td>
<td>1.1.1</td>
<td>Local Format</td>
<td>Arguments</td>
<td>1.5</td>
<td>7244</td>
<td>OSVDB-ID#12346</td>
</tr>
<tr>
<td>aspell</td>
<td>0.50.5</td>
<td>Local Stack</td>
<td>Local File</td>
<td>15.2</td>
<td>550</td>
<td>CVE-2004-0548</td>
</tr>
<tr>
<td>exim</td>
<td>4.41</td>
<td>Local Stack</td>
<td>Arguments</td>
<td>33.8</td>
<td>241856</td>
<td>EDB-ID#796</td>
</tr>
<tr>
<td>xserver</td>
<td>0.1a</td>
<td>Remote Stack</td>
<td>Sockets</td>
<td>31.9</td>
<td>1077</td>
<td>CVE-2007-3957</td>
</tr>
<tr>
<td>rsync</td>
<td>2.5.7</td>
<td>Local Stack</td>
<td>Env. Var</td>
<td>19.7</td>
<td>67744</td>
<td>CVE-2004-2093</td>
</tr>
<tr>
<td>xmail</td>
<td>1.21</td>
<td>Local Stack</td>
<td>Local File</td>
<td>1276.0</td>
<td>1766</td>
<td>CVE-2005-2943</td>
</tr>
<tr>
<td>corehttp</td>
<td>0.5.3</td>
<td>Remote Stack</td>
<td>Sockets</td>
<td>83.6</td>
<td>4873</td>
<td>CVE-2007-4060</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>114.6</strong></td>
<td><strong>56784</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **Overview**
- **AEG**
- **Conclusion**
Outline

1 Overview
   - Introduction
   - Considerations

2 AEG
   - Challenges
   - iwconfig Example
   - Constraint Equation
   - Components
   - Preconditioned Symbolic Execution

3 Conclusion
Future Work

- Consider heap management structures
- Integer overflow
- Timing attacks
- Platform independent and portable executables
- Binary only analysis
Effectiveness

- The average time to find a vulnerability was under 2 minutes
- Has been used in a timed competition - team had a working exploit in only a minute