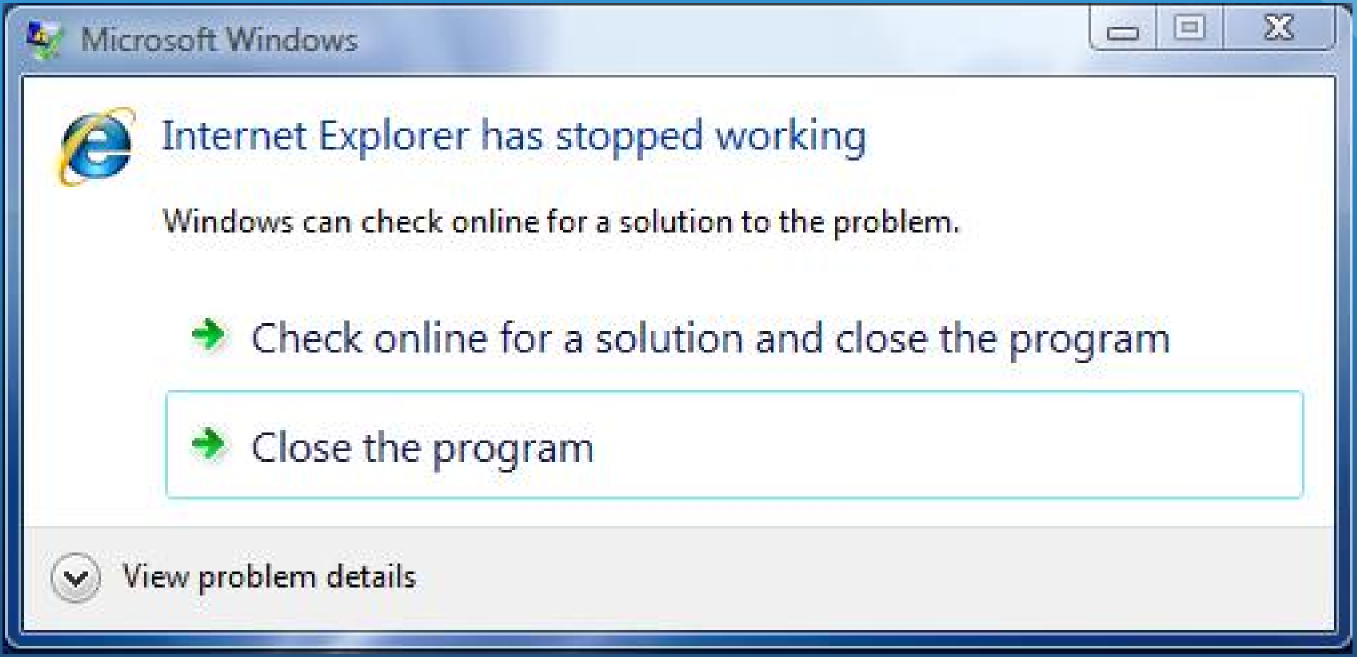
Better Bug Reporting with Better Privacy

M.Castro, M.Costa, JP. Martin

Presented by Horatiu Jula

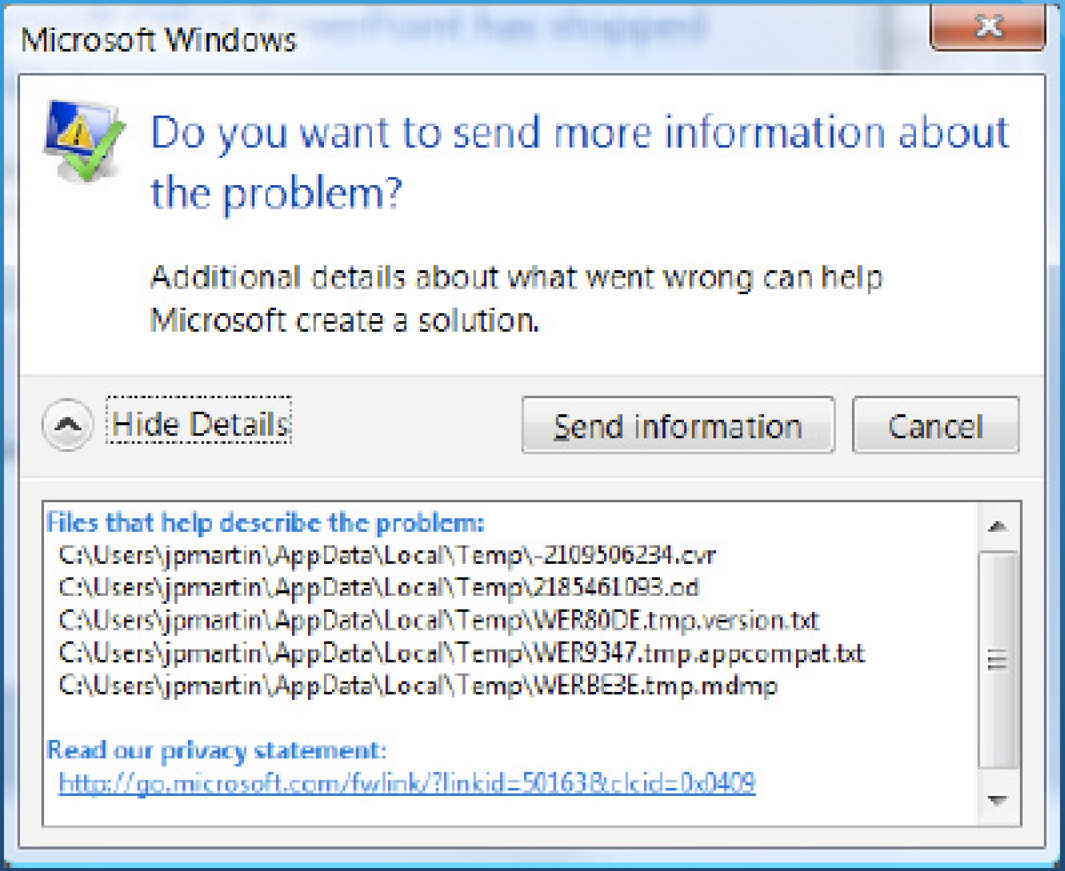
Imagine a crash



Internet Explorer has stopped working

Windows can check online for a solution to the problem.

Report the crash



Do you want to send more information about the problem?

Additional details about what went wrong can help Microsoft create a solution.

Files that help describe the problem:

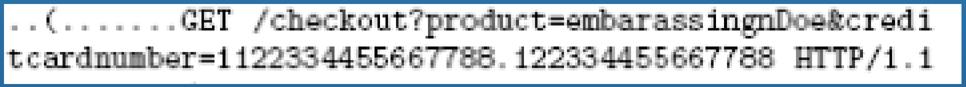
```
C:\Users\jpmartin\AppData\Local\Temp\-2109506234.cvr
C:\Users\jpmartin\AppData\Local\Temp\2185461093.od
C:\Users\jpmartin\AppData\Local\Temp\WER80DE.tmp.version.txt
C:\Users\jpmartin\AppData\Local\Temp\WER9347.tmp.appcompat.txt
C:\Users\jpmartin\AppData\Local\Temp\WERBE3E.tmp.mdmp
```

Read our privacy statement:

http://go.microsoft.com/fwlink/?linkid=50163&clcid=0x0409

Bug reporting today

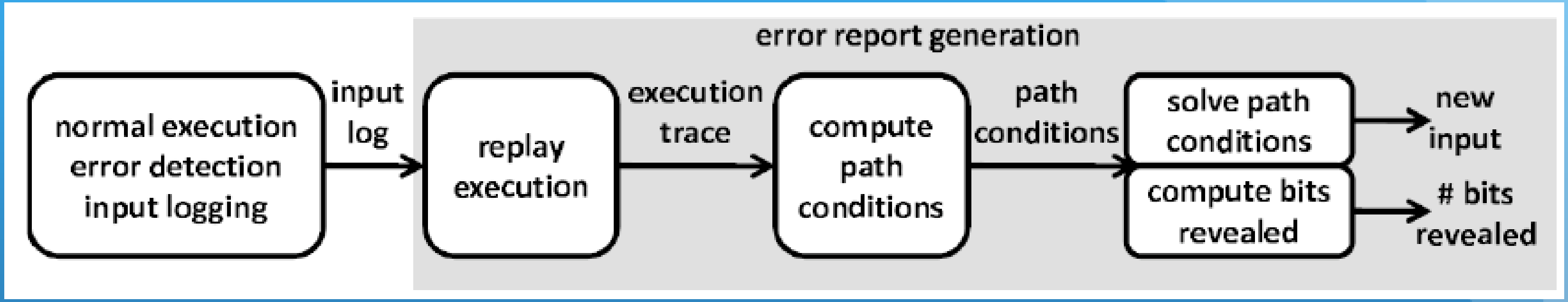
- Stack trace, memory dumps
  - May be insufficient
  - Solution: send path conditions
- Application-specific extras, failure-inducing document
  - May reveal private information



```
..(.......GET /checkout?product=embarassingnDoe&credi
tcardnumber=1122334455667788.122334455667788 HTTP/1.1
```

  - Users may not know if what they send contains private data
  - Solution: send a new document, without private data (if possible), that reveals the same bug

The approach



Error detection in normal execution → Input log

Replay bug in background → Instruction-level trace

Symbolically execute the trace → Path conditions that hold for the bad input and cause the bug

Solve the constraints to get

new inputs that satisfy the path conditions

#bits revealed from the original inputs

Example

```
int ProcessMessage(int sock, char *msg) {
    char url[20];
    char host[20];
    int i=0;
    if (msg[0] != ’G’ || msg[1] != ’E’
        || msg[2] != ’T’ || msg[3] != ’ ’)
        return -1;
    msg = msg+4;
    while (*msg != ’\n’ && *msg != ’ ’) {
        url[i++] = *msg++;
    }
    url[i] = 0;
    GetHost(msg, host);
    return ProcessGet(sock, url, host);
}
```

Buffer overflow, for i >= 20

Compute path conditions

```
int ProcessMessage(int sock, char *msg) {
        char url[20];
        char host[20];
        int i=0;
        if (msg[0] != ’G’ || msg[1] != ’E’
                || msg[2] != ’T’ || msg[3] != ’ ’)
                return -1;
        msg = msg+4;
        while (*msg != ’\n’ && *msg != ’ ’) {
                url[i++] = *msg++;
        }
        url[i] = 0;
        GetHost(msg, host);
        return ProcessGet(sock, url, host);
}
```

State:
*msg = b0,b1,b2,…
i = 0

Conditions:

Compute path conditions

```
int ProcessMessage(int sock, char *msg) {
    char url[20];
    char host[20];
    int i=0;
    if (msg[0] != ’G’ || msg[1] != ’E’
        || msg[2] != ’T’ || msg[3] != ’ ’)
        return -1;
➡   msg = msg+4;
    while (*msg != ’\n’ && *msg != ’ ’) {
        url[i++] = *msg++;
    }
    url[i] = 0;
    GetHost(msg, host);
    return ProcessGet(sock, url, host);
}
```

State:
*msg = b0,b1,b2,…
i = 0

Conditions:
b0=‘G’ /\ b1=‘E’ /\
b2=‘T’ /\ b3=‘ ’

Compute path conditions

```
int ProcessMessage(int sock, char *msg) {
    char url[20];
    char host[20];
    int i=0;
    if (msg[0] != ’G’ || msg[1] != ’E’
        || msg[2] != ’T’ || msg[3] != ’ ’)
        return -1;
    msg = msg+4;
    while (*msg != ’\n’ && *msg != ’ ’) {
        url[i++] = *msg++;
    }
    url[i] = 0;
    GetHost(msg, host);
    return ProcessGet(sock, url, host);
}
```

State:
*msg = b4,b5,b6,…
i = 0

Conditions:
b0=‘G’ /\ b1=‘E’ /\
b2=‘T’ /\ b3=‘ ’

Compute path conditions

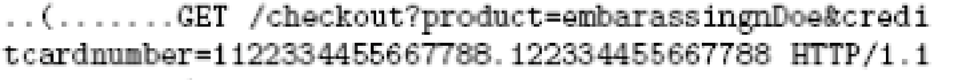
```
int ProcessMessage(int sock, char *msg) {
      char url[20];
      char host[20];
      int i=0;
      if (msg[0] != ’G’ || msg[1] != ’E’
            || msg[2] != ’T’ || msg[3] != ’ ’)
            return -1;
      msg = msg+4;
      while (*msg != ’\n’ && *msg != ’ ’) {
➡           url[i++] = *msg++;
      }
      url[i] = 0;
      GetHost(msg, host);
      return ProcessGet(sock, url, host);
}
```

State:
*msg = b20,b21,b22,…
*url = b4,b5,b6,…
i = 20

Conditions:
b0=‘G’ /\ b1=‘E’ /\
b2=‘T’ /\ b3=‘ ’ /\
b4 != ‘\n’ /\ b4 != ‘ ‘ /\
…
b20 != ‘\n’ /\ b20 != ‘ ‘

Summary

- Symbolic execution reveals the constraints under which a bug can occur
- Solving gives new inputs that trigger the same bug
- For our example
  - Memory dumps may reveal private information



```
..(.......GET /checkout?product=embarassingnDoe&credi
tcardnumber=1122334455667788.1223344556677 88 HTTP/1.1
```

  - New input: ‘GET ....’ (‘.’ represents byte value 0)
  - Only 4 bytes were relevant for the bug and had to be revealed

Evaluation

- Efficient technique
  - Generates reports quickly (<2min)
- Provides good privacy
  - Reveals very little of the original document (<15%)

Related work

- Vigilante (SOSP 2005)
  - Compute path conditions for an exploit and inline them into the application, as a filter for protecting the application against the exploit
- Bouncer (SOSP 2007)
  - Extends Vigilante with
    - Simplifying the path conditions
    - Learning new exploits by removing/duplicating bytes in the original exploit
      - New path conditions are derived for each new exploit
      - The final filter is a disjunction of the path conditions of the exploits