Microsoft's IE Is the Most Targeted Application By Security Researchers

Unknown Lamer posted yesterday | from darthcamaro

darthcamaro writes "Though Microsoft hasn't yet patched its Internet Explorer web browser in 2014, it did patch IE at least once every month in 2013. According to HP's 2013 Cyber Risk Report, more researchers tried to sell IE vulnerabilities than any other product vulnerability. 'IE is the most prevalent browser on the systems that attackers want to compromise' said Jacob West, CTO of HP's Enterprise Security Group."
# Integer overflows

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3)
    {
        return -1;
    }

    i = atoi(argv[1]);
    s = i;

    if(s >= 80) {
        /*[w1]*/
        printf("Oh no you don't!
");  
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

`nova:signed {100} ./width1 5 hello`
`s = 5
hello`

`nova:signed {101} ./width1 80 hello`
`Oh no you don't!`

`nova:signed {102} ./width1 65536 hello`
`s = 0
Segmentation fault (core dumped)`
Heap overflows

Low memory addresses ↔ High memory addresses

.env.

.text .data .bss heap stack

Low memory addresses

attacker buffer attacker ptr

High memory addresses
Format-string vulnerabilities

```c
#define A_acker controls format string gives all sorts of control

void main(int argc, char* argv[]) {
    printf( argv[1] );
}
```

argv[1] = “%ss%s%s%s%s%s%s%s%s%s”

Attacker controls format string gives all sorts of control

Can do control hijacking directly
1.1 Buffer Overflows vs. Format String Vulnerabilities

As with every vulnerability, it was developed over time and new techniques have shown up, often because old ones did not work in a certain situation. People who truly deserve credit for a lot of techniques mentioned in this article and have influenced my writing significantly are:
- tf8
- portal
- DiGiT
- smile

Although I have contributed some tricks myself, without the giant help of comments and tricks both theoretically or in form of an exploit, shown to me by these people, this article would not have been possible. Thanks.

I also thank the numerous individuals who commented, reviewed, and improved this article.

Updated and corrected versions may appear on the TESO Security Group homepage.

### 1.1 Buffer Overflows vs. Format String Vulnerabilities

<table>
<thead>
<tr>
<th></th>
<th>Buffer Overflow</th>
<th>Format String</th>
</tr>
</thead>
<tbody>
<tr>
<td>public since</td>
<td>mid 1980’s</td>
<td>June 1999</td>
</tr>
<tr>
<td>danger realized</td>
<td>1990’s</td>
<td>June 2000</td>
</tr>
<tr>
<td>number of exploits</td>
<td>a few thousand</td>
<td>a few dozen</td>
</tr>
<tr>
<td>considered as</td>
<td>security threat</td>
<td>programming bug</td>
</tr>
<tr>
<td>techniques</td>
<td>evolved and advanced</td>
<td>basic techniques</td>
</tr>
<tr>
<td>visibility</td>
<td>sometimes very difficult to spot</td>
<td>easy to find</td>
</tr>
</tbody>
</table>

From “Exploiting format string vulnerabilities”
Summary

• Classic buffer overflow
  – corrupt program control data
  – hijack control flow easily
• Integer overflow, signedness, format string, heap overflow, ...
• These were all local privilege escalation vulns
  – Similar concepts for remote vulnerabilities
• Defenses?
Finding vulnerabilities

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
Finding vulnerabilities

Manual analysis

Simple example: double free

Fuzzing tools

Static analysis, dynamic analysis

...
Hackers use People, Processes and Technology to obtain a singular goal: Information dominance

From “How Hackers Look for Bugs”, Dave Aitel
Take a sample product X and attack it remotely

From “How Hackers Look for Bugs”, Dave Aitel
Program analyzers

analyze large code bases

potentially reports many warnings

may emit false alarms

**Report** | **Type**    | **Line**
---|-------------|-------
1  | mem leak    | 324   
2  | buffer oflow | 4,353,245 
3  | sql injection | 23,212 
4  | stack oflow | 86,923 
5  | dang ptr    | 8,491 
... | ...         | ...  
10,502 | info leak  | 10,921
Example program analyzers

• Manual analysis (you are the analyzer!)
• Static analysis (do not execute program)
  – Scanners
  – Abstract interpretation
  – Symbolic execution
• Dynamic analysis (execute program)
  – Debugging
  – Fuzzers
  – Ptrace

Do you have source code?
Yes: lucky you
No: can still do things, but not as easily
  (missing a lot of context about program)
### Program analysis: Soundness and completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Soundness</strong></td>
<td>If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”</td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”</td>
</tr>
<tr>
<td>Complete</td>
<td>Incomplete</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>Sound</strong></td>
<td><strong>Undecidable</strong></td>
</tr>
<tr>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td>No false positives</td>
<td>No false negatives</td>
</tr>
<tr>
<td>No false negatives</td>
<td>False positives</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Unsound</strong></th>
<th><strong>Decidable</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td>False positives</td>
<td>False negatives</td>
</tr>
<tr>
<td>No false negatives</td>
<td>False positives</td>
</tr>
</tbody>
</table>

---

Slide credit: Prof Mitchell Stanford’s CS 155
Manual analysis

• You get a binary or the source code
• You find vulnerabilities

• Experienced analysts according to Aitel:
  – 1 hour of binary analysis:
    • Simple backdoors, coding style, bad API calls (strcpy)
  – 1 week of binary analysis:
    • Likely to find 1 good vulnerability
  – 1 month of binary analysis:
    • Likely to find 1 vulnerability no one else will ever find
Disassembly and decompiling

The normal compilation process

What if we start with binary?

Disassembler
(gdb, IDA Pro, OllyDebug)

Decompiler
(IDA Pro has one)

Very complex, usually poor results
Tool example: IDA Pro
Tool example: IDA Pro
What type of vulnerability might this be?

```
main( int argc, char* argv[] ) {
    char* b1;
    char* b2;
    char* b3;

    if( argc != 3 ) then return 0;
    if( argv[2] != 31337 )
        complicatedFunction();
    else {
        b1 = (char*)malloc(248);
        b2 = (char*)malloc(248);
        free(b1);
        free(b2);
        b3 = (char*)malloc(512);
        strncpy( b3, argv[1], 511 );
        free(b2);
        free(b3);
    }
}
```

Double-free vulnerability
Double-free vulnerabilities

Can corrupt the state of the heap management

Say we use a simple doubly-linked list malloc implementation with control information stored alongside data

<table>
<thead>
<tr>
<th>chunk.left</th>
<th>fb</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk.right</td>
<td></td>
</tr>
<tr>
<td>user data</td>
<td></td>
</tr>
</tbody>
</table>

Chunk has:
1) left ptr (to previous chunk)
2) right ptr (to next chunk)
3) free bit which denotes if chunk is free this reuses low bit of right ptr because we will align chunks
4) user data
malloc()
- search left-to-right for free chunk
- modify pointers
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 );
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 )
b2 = malloc( BUF_SIZE2 )

free()
- Consolidate with free neighbors
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 )
b2 = malloc( BUF_SIZE2 )

free()
- Consolidate with free neighbors

free( b1 )
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 )
b2 = malloc( BUF_SIZE2 )

free()
- Consolidate with free neighbors

free( b1 )
free( b2 )
malloc()
- search left-to-right for free chunk
- modify pointers

\[
b_1 = \text{malloc}(\ \text{BUF\_SIZE1}\ )
\]
\[
b_2 = \text{malloc}(\ \text{BUF\_SIZE2}\ )
\]

free()
- Consolidate with free neighbors

\[
\text{free}(\ b_1\ )
\]
\[
\text{free}(\ b_2\ )
\]
\[
\text{free}(\ b_2\ )
\]
\[
b_3 = \text{malloc}(\ \text{BUF\_SIZE1} + \text{BUF\_SIZE2}\ )
\]
malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc( BUF_SIZE1 )
b2 = malloc( BUF_SIZE2 )

free()
- Consolidate with free neighbors

d1 = free( b1 )
d2 = free( b2 )
b3 = malloc( BUF_SIZE1 + BUF_SIZE2 )
strncpy( b3, argv[1], BUF_SIZE1+BUF_SIZE2-1 )
malloc()
- search left-to-right for free chunk
- modify pointers

\[
b_1 = \text{malloc}( \text{BUF\_SIZE1} )
\]
\[
b_2 = \text{malloc}( \text{BUF\_SIZE2} )
\]

free()
- Consolidate with free neighbors

\[
\text{free}( b_1 )
\]
\[
\text{free}( b_2 )
\]
\[
\text{free}( b_2 )
\]
\[
\text{strncpy}( b_3, \text{argv}[1], \text{BUF\_SIZE1}+\text{BUF\_SIZE2}-1 )
\]
\[
\text{free}( b_2 )
\]

With a clever argv[1]:
write a 4-byte word to an arbitrary location in memory

\[
(b_2 - 8)\rightarrow\text{left}\rightarrow\text{right} = (b_2-8)\rightarrow\text{right}
\]
\[
(b_2 - 8)\rightarrow\text{right}\rightarrow\text{left} = (b_2-8)\rightarrow\text{left}
\]
What type of vulnerability might this be?

This is very simple example. Manual analysis is very time consuming.

Security analysts use a variety of tools to augment manual analysis.
Aiding analysts with tools

How can we automatically find the bug?

```c
main( int argc, char* argv[] ) {
  char* b1;
  char* b2;
  char* b3;

  if( argc != 3 ) then return 0;
  if( argv[2] != 31337 )
    complicatedFunction();
  else {
    b1 = (char*)malloc(248);
    b2 = (char*)malloc(248);
    free(b1);
    free(b2);
    b3 = (char*)malloc(512);
    strncpy( b3, argv[1], 511 );
    free(b2);
    free(b3);
  }
}
```
Start with dynamic analysis: Fuzzing

“The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.”

Wikipedia
http://en.wikipedia.org/wiki/Fuzz_testing

Choose a bunch of inputs
See if they cause program to misbehave
Example of dynamic analysis
Black-box fuzz testing: the goal

Normal input → Program → output(s)

Mutated input → Program → output(s)

Mutated input 2 → Program → Program crash
Black-box fuzz testing

If \( x \) is 32 bits, then probability of crashing is \textbf{at most what?} \( \frac{1}{2^{32}} \)

Achieving code coverage can be very difficult

```
main( int argc, char* argv[] ) {
    char* b1;
    char* b2;
    char* b3;

    if( argc != 3 ) return 0;
    if( argv[2] != 31337 )
        complicatedFunction();
    else {
        b1 = (char*)malloc(248);
        b2 = (char*)malloc(248);
        free(b1);
        free(b2);
        b3 = (char*)malloc(512);
        strncpy( b3, argv[1], 511 );
        free(b2);
        free(b3);
    }
}
```
Fuzzing is a lot about code coverage

• Code coverage defined in many ways
  – # of basic blocks reached
  – # of paths followed
  – # of conditionals followed
  – gcov is useful standard tool

• Mutation based
  – Start with known-good examples
  – Mutate them to new test cases
    • heuristics: increase string lengths (AAAAAAAAAAA...)
    • randomly change items

• Generative
  – Start with specification of protocol, file format
  – Build test case files from it
    • Rarely used parts of spec
Manually refine fuzzing
(example from Miller slides)

Multiplayer game
Fuzz for remote exploits
• Capture packets during normal use
• Replace some packet contents with random values
• Send to game, determine code coverage

Initial: 614 out of 36183 basic blocks

One big switch statement controlled by third byte of packet
Update fuzz rules to exhaust the values of this third byte

Improves coverage by 4x.
Repeat several times to improve coverage.
Heap overflow found.

From Wikipedia:
Example program analyzers

• Manual analysis (you are the analyzer!)
• Static analysis (do not execute program)
  – Scanners
  – Symbolic execution
  – Abstract representations
• Dynamic analysis (execute program)
  – Debugging
  – Fuzzers
  – Ptrace

Do you have source code?
Yes: lucky you
No: can still do things, but not as easily
(missing a lot of context about program)
Source code scanners

Look at source code, flag suspicious constructs

```
... strcpy(ptr1, ptr2);
...
```

Warning: Don’t use `strcpy`

Simplest example: `grep`
Lint is early example
RATS (Rough auditing tool for security)
ITS4 (It’s the Software Stupid Security Scanner)

Circa 1990’s technology:

*shouldn’t* work for reasonable modern codebases
Symbolic execution

- Technique for statically analyzing code paths and finding inputs
- Associate to each input variable a special symbol
  - called symbolic variable
- Simulate execution symbolically
  - Update symbolic variable’s value appropriately
  - Conditionals add constraints on possible values
- Cast constraints as satisfiability, and use SAT solver to find inputs

Source Code → Clang → LLVM bitcode → Symbolic Executor (e.g., KLEE)

Symbolic Executor

Bug found
Input that gets to that bug
main( int argc, char* argv[] ) {
  char* b1;
  char* b2;
  char* b3;

  if( argc != 3 ) then return 0;
  if( argv[2] != 31337 ) complicatedFunction();
  else {
    b1 = (char*)malloc(248);
    b2 = (char*)malloc(248);
    free(b1);
    free(b2);
    b3 = (char*)malloc(512);
    strncpy( b3, argv[1], 511 );
    free(b2);
    free(b3);
  }
}

Initially:
argc = x (unconstrained int)
argv[2] = z (memory array)

- Eventually emulation hits a double free
- Can trace back up path to determine what x, z must have been to hit this basic block
Symbolic execution challenges

• Can we complete analyses?
  – Yes, but only for very simple programs
  – Exponential # of paths to explore

• Path selection
  – Might get stuck in complicatedFunction()

• Encoding checks on symbolic states
  – Must include logic for double free check
  – Symbolic execution on binary more challenging (lose most memory semantics)
Software

Reported Error

False Alarm

Sound Over-approximation of Behaviors

approximation is too coarse...
...yields too many false alarms

Slide credit: Prof Mitchell Stanford’s CS 155
Does this program ever crash?

Slide credit: Prof Mitchell Stanford’s CS 155
Does this program ever crash?

Infeasible path!
... program will never crash

Slide credit: Prof Mitchell Stanford’s CS 155
Try analyzing without approximating...

non-termination!
... therefore, need to approximate

Slide credit: Prof Mitchell Stanford’s CS 155
Try analyzing with “signs” approximation...

entry

\(X \leftarrow 0\)

Is \(Y = 0\) ?

\(X = 0\)

\(X = 0\)

\(X = X + 1\)

\(X = X - 1\)

Is \(Y = 0\) ?

\(X = pos\)

\(X = neg\)

\(X = T\)

\(X = T\)

Is \(X < 0\) ?

\(X = T\)

\(X = T\)

crash

exit

... but reports false alarm
... therefore, need more precision

Slide credit: Prof Mitchell Stanford’s CS 155
Try analyzing with “path-sensitive signs” approximation...

entry

\[ X \leftarrow 0 \]

Is \( Y = 0 \)?

\begin{align*}
Y = 0 & \quad X = 0 \\
Y \neq 0 & \quad X = \text{pos} \\
\end{align*}

\begin{align*}
Y = 0 & \quad X = \text{pos} \\
Y \neq 0 & \quad X = \text{neg} \\
\end{align*}

\begin{align*}
Y = 0 & \quad X = \text{pos} \\
Y \neq 0 & \quad X = \text{neg} \\
\end{align*}

\begin{align*}
Y = 0 & \quad X = \text{pos} \\
Y \neq 0 & \quad X = \text{neg} \\
\end{align*}

\begin{align*}
Y = 0 & \quad X = \text{pos} \\
Y \neq 0 & \quad X = \text{neg} \\
\end{align*}

terminates...

... no false alarm

... soundly proved never crashes

Slide credit: Prof Mitchell Stanford’s CS 155
We represent program states using lattices; this specifies the possible values we assign to variables.
Bug finding is a big business

• Grammatech (Prof Reps here at Wisconsin)
• Coverity (Stanford startup)
• Fortify
• many, many others…
Example program analyzers

• Manual analysis (you are the analyzer!)
• Static analysis (do not execute program)
  – Scanners
  – Abstract interpretation
  – Symbolic execution
• Dynamic analysis (execute program)
  – Debugging
  – Fuzzers
  – Ptrace

Do you have source code?
Yes: lucky you
No: can still do things, but not as easily
  (missing a lot of context about program)
"The most critical servers contain malicious software that can normally be detected by anti-virus software," it says. "The separation of critical components was not functioning or was not in place. We have strong indications that the CA-servers, although physically very securely placed in a tempest proof environment, were accessible over the network from the management LAN."

All CA servers were members of one Windows domain and all accessible with one user/password combination. Moreover, the used password was simple and susceptible to brute-force attacks.

http://www.net-security.org/secworld.php?id=11570
Taint tracking

Track information flow from user input to it’s use

Can be either static or dynamic

Useful to augment manual testing

```c
strcpy(buf, argv[1]);
```
White-box fuzz testing

• Start with real input and do static analysis
  – Symbolic execution of program
  – Gather constraints (control flow) along way
  – Systematically negate constraints backwards
  – Eventually this yields a new input

• Repeat

Godefroid, Levin, Molnar. “Automated Whitebox Fuzz Testing”
Symbolic execution + fuzzing

void top(char input[4]) {
  int cnt = 0;
  if (input[0] == ’b’) cnt++;
  if (input[1] == ’a’) cnt++;
  if (input[2] == ’d’) cnt++;
  if (input[3] == ’!’) cnt++;
  if (cnt >= 3) abort(); // error
}

Example from Godefroid et al.

Start with some input.
Run program for real & symbolically
Say input = “good”

i0 != ‘b’
i1 != ‘a’
i2 != ‘d’
i3 != ‘!’

This gives set of constraints on input
Negate them one at a time to generate a new input that explores new path

Example
i0 != ‘b’ and i1 != ‘a’ and i2 != ‘d’ and i3 = ‘!’
input would be ``goo!’’

Repeat with new input
Figure 2. Search space for the example of Figure 1 with the value of the variable \texttt{cnt} at the end of each run and the corresponding input string.

Example from Godefroid et al.

Larger programs have too many paths to explore so they specify various heuristics

In-use at Microsoft
$X = 0$

$X \leftarrow X + 1$

$X = 1$

$d_{\text{out}} = f(d_{\text{in}})$

Dataflow elements

Transfer function

Dataflow equation

Slide credit: Prof Mitchell Stanford’s CS 155
\( X = 0 \)  

\[ X \leftarrow X + 1 \]

\( X = 1 \)

\[ X \leftarrow X + 1 \]

\( \text{Is } Y = 0 ? \)

\( X = 1 \)

\[ X \leftarrow X + 1 \]

\( X = 1 \)

\( X = 1 \)

\( X = 1 \)

\( X = 0 \)

\[ f_1(d_{\text{in}1}) \]

\[ d_{\text{out}1} = f_1(d_{\text{in}1}) \]

\( d_{\text{out}1} = d_{\text{in}2} \)

\[ d_{\text{out}2} = f_2(d_{\text{in}2}) \]

\( d_{\text{out}2} = f_2(d_{\text{in}2}) \)

Slide credit: Prof Mitchell Stanford’s CS 155
What is the space of dataflow elements, \( \Delta \)?
What is the least upper bound operator, \( \sqcup \)?

\[
d_{out1} = f_1(d_{in1})
\]
\[
d_{out2} = f_2(d_{in2})
\]
\[
d_{join} = d_{out1} \sqcup d_{out2}
\]
\[
d_{join} = d_{in3}
\]
\[
d_{out3} = f_3(d_{in3})
\]

least upper bound operator
Example: union of possible values

Slide credit: Prof Mitchell Stanford’s CS 155
We give a lattice to specify the possible values we assign to symbolic variables
Try analyzing with “signs” approximation...

```
Is \( X \leq 0 \) ?
```

```
X \leftarrow 0
```

```
X \leftarrow X + 1
```

```
X \leftarrow X - 1
```

```
Is Y = 0 ?
```

```
X = 0
```

```
X = pos
```

```
X = T
```

```
X = neg
```

```
X = T
```

```
Is X < 0 ?
```

```
Is Y = 0 ?
```

```
exit
```

```
crash
```

```
X = T
```

```
X = T
```

```
X = T
```

```
X = T
```

```
X = T
```

```
X = T
```

```
X = T
```

lost precision

terminates...
... but reports false alarm
... therefore, need more precision

Slide credit: Prof Mitchell Stanford’s CS 155
Try analyzing with “path-sensitive signs” approximation...

- **entry**
  - **X ← 0**
  - **Is Y = 0?**
    - **yes**
      - **Y = 0**
        - **X = 0**
        - **true**
        - **no**
          - **X = 0**
          - **Y ≠ 0**

    - **no**
      - **X = X + 1**
      - **X = X - 1**
        - **Is Y = 0?**
          - **yes**
            - **Y = 0**
              - **X = pos**
              - **true**
              - **false**
            - **X = neg**
              - **Y ≠ 0**
          - **no**
            - **X = neg**
            - **Y ≠ 0**

- **refinement**
  - **Y = 0**
    - **X = pos**
    - **true**
    - **false**
  - **Y ≠ 0**
    - **X = neg**
    - **true**
    - **false**

- **terminates...**
  - **no false alarm**
  - **soundly proved never crashes**

Slide credit: Prof Mitchell Stanford’s CS 155