Data-Oriented Programming

On the Expressiveness of Non-Control Data Attacks

Hong Hu, Shweta Shinde,
Sendroiu Adrian, Zheng Leong Chua,
Prateek Saxena, Zhenkai Liang

Department of Computer Science
National University of Singapore
Control Attacks are Getting Harder
Control Attacks are Getting Harder
Control Attacks are getting harder

- Code injection

CFG

memory space

- Data
- Code
Control Attacks are Getting Harder

- Code injection ← Data Execution Prevention
Control Attacks are Getting Harder

- Code injection
  - Data Execution Prevention
- Code reuse
  - return-to-libc
  - return-oriented programming (ROP)
Control Attacks are Getting Harder

- Code injection
- Code reuse
  - return-to-libc
  - return-oriented programming (ROP)

- Data Execution Prevention
- Control Flow Integrity

CFG w/ CFI

Memory space

Data w/ DEP

Code
A New Attack Class

- Assume: conform to CFI & DEP
A New Attack Class

• Assume: conform to CFI & DEP
• Attackers’ capability on arbitrary vul. programs?

Nothing  Specific computation  Turing-complete

CFG w/ CFI

Memory space
Data w/ DEP
Code
Non-Control Data Attacks

• Corrupt/leak several bytes of security-critical data
Non-Control Data Attacks

- Corrupt/leak several bytes of security-critical data

```c
//set root privilege
seteuid(0);
......
//set normal user privilege
seteuid(pw->pw_uid);
//execute user’s command

//offset depends on IE version
safemode = *(DWORD *) (jsobj + offset);
if(safemode & 0xB == 0) {
    Turn_on_God_Mode();
}
```

+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014
Non-Control Data Attacks

• Corrupt/leak several bytes of security-critical data

```c
//set root privilege
seteuid(0);
......
//set normal user privilege
seteuid(pw->pw_uid);
//execute user’s command
```

```c
//offset depends on IE version
safemode = *(DWORD *)(jsobj + offset);
if(safemode & 0xB == 0) {
    Turn_on_God_Mode();
}
```

• Special cases relying on particular data/functions
  – user id, safemode, private key, etc
  – interpreter – printf() (with “%n”), etc

Nothing  Specific computation  Turing-complete

+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014
Contributions

• Non-control data attacks can be Turing-complete
Contributions

• Non-control data attacks can be Turing-complete

• Data-Oriented Programming (DOP)
  – build expressive non-control data attacks
  – independent of any specific data / functions
Contributions

• Non-control data attacks can be Turing-complete

• Data-Oriented Programming (DOP)
  – build expressive non-control data attacks
  – independent of any specific data / functions

• DOP builds attacks on real-world programs
  – bypass ASLR w/o address leakage
  – simulate a network bot
  – enable code injection
Motivating Example

Vulnerable Program

```c
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
    readData(sockfd, buf); // stack bof
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    } //...(following code skipped)...
}
```
Motivating Example

```c
struct server{int *cur_max, total, typ;}; *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
    readData(sockfd, buf);    // stack bof
    if(*type == NONE) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
    //...(following code skipped)...
}
```

Vulnerable Program
Motivating Example

Vulnerable Program

```c
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12];
...
while (quota--) {
  readData(sockfd, buf); // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
    *size = *(srv->cur_max);
  else {
    srv->typ = *type;
    srv->total += *size;
  } //...(following code skipped)...
} //...(following code skipped)...
```
Motivating Example

```
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12];
...
while (quota--) {
  readData(sockfd, buf); // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
    *size = *(srv->cur_max);
  else {
    srv->typ = *type;
    srv->total += *size;
  } //...(following code skipped)...
}
```

Vulnerable Program

```
struct Obj {struct Obj *next; int prop;}
void updateList(struct Obj *list, int addend){
  for(; list != NULL; list = list->next)
    list->prop += addend;
}
```

Malicious Computation
Motivating Example

1 struct server{int *cur_max, total, typ;} *srv;
2 int quota = MAXCONN; int *size, *type;
3 char buf[MAXLEN];
4 size = &buf[8]; type = &buf[12]
5 ...
6 while (quota--) {
7    readData(sockfd, buf); // stack bof
8    if(*type == NONE ) break;
9    if(*type == STREAM)
10       *size = *(srv->cur_max);
11    else {
12       srv->typ = *type;
13       srv->total += *size;
14    } //...(following code skipped)...  
15 }

Vulnerable Program

Malicious Computation

CFG w/ CFI
Motivating Example

struct server { int *cur_max, total, typ; } *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
    readData(sockfd, buf); // stack bof
    if (*type == NONE ) break;
    if (*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    } //...(following code skipped)...
}

1 struct Obj {struct Obj *next; int prop;}
2 void updateList(struct Obj *list, int addend){
    for (; list != NULL; list = list->next)
        list->prop += addend;
}
Motivating Example

```c
struct server { int *cur_max, total, typ; } *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
    readData(sockfd, buf); // stack bof
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    } //...(following code skipped)...
}
```

Vulnerable Program

```
struct Obj { structure Obj *next; int prop;}

void updateList(struct Obj *list, int addend) {
    for (; list != NULL; list = list->next)
        list->prop += addend;
}
```

Malicious Computation
Motivating Example (cont.)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>while (quota--) {</td>
</tr>
<tr>
<td>7</td>
<td>readData(sockfd, buf);</td>
</tr>
<tr>
<td>8</td>
<td>if(*type == NONE ) break;</td>
</tr>
<tr>
<td>9</td>
<td>if(*type == STREAM)</td>
</tr>
<tr>
<td>10</td>
<td>*size = *(srv-&gt;cur_max);</td>
</tr>
<tr>
<td>11</td>
<td>else {</td>
</tr>
<tr>
<td>12</td>
<td>srv-&gt;typ = *type;</td>
</tr>
<tr>
<td>13</td>
<td>srv-&gt;total += *size;</td>
</tr>
<tr>
<td>14</td>
<td>}</td>
</tr>
<tr>
<td>15</td>
<td>} vulnerable program</td>
</tr>
<tr>
<td>4</td>
<td>for(; list != NULL; list = list-&gt;next)</td>
</tr>
<tr>
<td>5</td>
<td>list-&gt;prop += addend;</td>
</tr>
</tbody>
</table>

malicious computation

simulate ?

simulate ?
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```
for(; list != NULL; list = list->next) {
    list->prop += addend;
}
```

`malicious computation`

`simulate` ?

---

**Memory space**

```
buf[]  type  size  quota  srv
```

```
cur_max  total  typ
```

---

24 **Memory space**

vulnerable program
Motivating Example (cont.)

```c
4  for(; list != NULL; list = list->next) {
5      list->prop += addend;
}
```

```
6  while (quota--) {
7      readData(sockfd, buf);
8      if(*type == NONE ) break;
9      if(*type == STREAM) {
10         *size = *(srv->cur_max);
11      } else {
12         srv->typ = *type;
13         srv->total += *size;
14      }
15  }
```

Memory space

```
buf[]    type    size    quota    srv
```

```
heap
```

```
next    prop
```

```
stack
```

```
cur_max    total    typ
```

```
addend
```

```
list
```

```
```
malicious computation
simulate
vulnerable program
```
Motivating Example (cont.)

```
5 while (quota--) {
6    readData(sockfd, buf);
7    if(*type == NONE ) break;
8    if(*type == STREAM)
9        *size = *(srv->cur_max);
10   else {
11      srv->typ = *type;
12      srv->total += *size;
13   }
14 }
```

```
4 for(; list != NULL; list = list->next) {
5    list->prop += addend;
```

malicious computation

simulate

vulnerable program

Memory space

```
buf[]  type  size  quota  srv
```

```
list
```

```
addend
```

```
next  prop
```

```
cur_max  total  typ
```

```
stack
```
Motivating Example (cont.)

```
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```
for(; list != NULL; list = list->next) {
    list->prop += addend;
}
```

Malicious computation simulates vulnerable program
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```
for(; list != NULL; list = list->next)
list->prop += addend;
```

Malicious computation

Simulate?

Memory space

```
buf[]  type  size  quota  srv
```

Stack

```
cur_max  total  typ
```

Heap

```
addend  list  next  prop
```

Vulnerable program
Motivating Example (cont.)

4 \textbf{for} (; \texttt{list} \neq \texttt{NULL}; \texttt{list} = \texttt{list}->\texttt{next})
5 \texttt{list}->\texttt{prop} += \texttt{addend};

malicious computation

simulate ?

Memory space

\begin{tabular}{|c|c|c|c|c|}
\hline
\texttt{buf[]} & \texttt{type} & \texttt{size} & \texttt{quota} & \texttt{srv} \\
\hline
\end{tabular}

\begin{itemize}
\item \texttt{heap}
\item \texttt{stack}
\end{itemize}

\begin{itemize}
\item \texttt{addend}
\item \texttt{list}
\item \texttt{next prop}
\item \texttt{cur_max} \texttt{total} \texttt{typ}
\item \texttt{vulnerable program}
\end{itemize}
Motivating Example (cont.)

```c
4  for(; list != NULL; list = list->next) 
5      list->prop += addend;
6  while (quota--) {
7      readData(sockfd, buf);
8      if(*type == NONE ) break;
9      if(*type == STREAM)
10         *size = *(srv->cur_max);
11      else {
12         srv->typ = *type;
13         srv->total += *size;
14      } 
15  }  
```

```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
    *size = *(srv->cur_max);
  else {
    srv->typ = *type;
    srv->total += *size;
  }
```

```
4  for(; list != NULL; list = list->next) 
5      list->prop += addend;
```
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}

for(; list != NULL; list = list->next) {
    list->prop += addend;
}

Motivating Example (cont.)

malicious computation

simulate

vulnerable program

Memory space

buf[]  type  size  quota  srv

stack

cur_max  total  typ

heap

list

next  prop

next  prop

addend
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```c
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation

simulate ?

Vulnerable program

Memory space

- buf[]
- type
- size
- quota
- srv

Stack

- cur_max
- total
- typ

Heap

- list
- addend
- next
- prop

```
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

```
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

vulnerable program

Malicious computation

simulate ?

Memory space

<table>
<thead>
<tr>
<th>buf[]</th>
<th>type</th>
<th>size</th>
<th>quota</th>
<th>srv</th>
</tr>
</thead>
</table>

stack

<table>
<thead>
<tr>
<th>cur_max</th>
<th>total</th>
<th>typ</th>
</tr>
</thead>
</table>

heap

<table>
<thead>
<tr>
<th>list</th>
</tr>
</thead>
</table>

addend

<table>
<thead>
<tr>
<th>cur_max</th>
<th>total</th>
<th>typ</th>
</tr>
</thead>
</table>

next | prop
---|---
next | prop
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}  // vulnerable program
```

```c
for(; list != NULL; list = list->next)
    list->prop += addend;
```
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}

while (list != NULL) {
    list->prop += addend;
    list = list->next;
}

Motivating Example (cont.)

malicious computation

simulate ?

heap

stack

Memory space

buf[]  type  size  quota  srv

list  addend

cur_max  total  typ

next  prop

next  prop
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

Malicious computation

Simulate ?

Memory space

![Diagram of memory space with stack and heap structures]
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

```
Motivating Example (cont.)
```

```
malicious computation
```

```
simulate
```

```
vulnerable program
```
Motivating Example (cont.)

6 \textbf{while} (quota--) {
7 \hspace{1em} \texttt{readData(sockfd, buf)};
8 \hspace{1em} \textbf{if}(*type == \texttt{NONE}) \textbf{break};;
9 \hspace{1em} \textbf{if}(*type == \texttt{STREAM})
10 \hspace{1em} \hspace{1em} *size = *(srv->cur\_max);
11 \hspace{1em} \textbf{else} {
12 \hspace{2em} srv->typ = *type;
13 \hspace{2em} srv->total += *size;
14 \hspace{1em} }
15 \}

\texttt{vulnerable program}

4 \textbf{for}(; list != NULL; list = list->next)
5 \hspace{1em} list->prop += addend;

\texttt{malicious computation}

\texttt{simulate}?
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

malicious computation

simulate ?

```c
for(; list != NULL; list = list->next)
list->prop += addend;
```
Motivating Example (cont.)

```c
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM) {
        *size = *(srv->cur_max);
    } else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

```c
for(; list != NULL; list = list->next)
list->prop += addend;
```

malicious computation

simulate ?

---

**Memory space**

- `buf[]`
- `type`
- `size`
- `quota`
- `srv`
- `stack`
  - `cur_max`
  - `total`
  - `typ`

---

**heap**

- `STREAM`
- `addend`
- `list`
- `next`
- `prop`
Motivating Example (cont.)

```c
4 for(; list != NULL; list = list->next) {
5    list->prop += addend;
}
```

```c
6 while (quota-- ) { 
7    readData(sockfd, buf);
8    if(*type == NONE ) break;
9    if(*type == STREAM) 
10        *size = *(srv->cur_max);
11    else { 
12        srv->typ = *type;
13        srv->total += *size;
14    }
15 } }
```

vulnerable program

Memory space

```
<table>
<thead>
<tr>
<th>buf[]</th>
<th>type</th>
<th>size</th>
<th>quota</th>
<th>srv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stack</td>
</tr>
</tbody>
</table>
```

heap

```
<table>
<thead>
<tr>
<th>cur_max</th>
<th>total</th>
<th>typ</th>
</tr>
</thead>
</table>
```

stack

```
<table>
<thead>
<tr>
<th>next</th>
<th>prop</th>
</tr>
</thead>
</table>
```

STREAM

malicious computation

simulate ✓
Data-Oriented Programming

A Generic Technique
Data-Oriented Programming (DOP)

• General construction
  – w/o dependency on specific data / functions
Data-Oriented Programming (DOP)

• General construction
  – w/o dependency on specific data / functions

• Expressive attacks
  – towards Turing-complete computation
Data-Oriented Programming (DOP)

- General construction
  - w/o dependency on specific data / functions
- Expressive attacks
  - towards Turing-complete computation
- Elements
  - data-oriented gadgets
  - gadget dispatchers
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)

Addition: `srv->total += *size;`

1. `mov (%esi), %ebx`  //load micro-op
2. `mov (%edi), %eax`  //load micro-op
3. `add %ebx, %eax`  //addition
4. `mov %eax, (%edi)`  //store micro-op
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)
  – save results in memory
  – load micro-op --> semantics
    micro-op --> store micro-op

Addition: srv->total += *size;

1  mov (%esi), %ebx   //load micro-op
2  mov (%edi), %eax   //load micro-op
3  add %ebx, %eax     //addition
4  mov %eax, (%edi)   //store micro-op
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)
  – save results in memory
  – load micro-op --> semantics
    micro-op --> store micro-op

Addition: `srv->total += *size;`

1. `mov (%esi), %ebx` //load micro-op
2. `mov (%edi), %eax` //load micro-op
3. `add %ebx, %eax` //addition
4. `mov %eax, (%edi)` //store micro-op
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)
  – save results in memory
  – load micro-op --> semantics
    micro-op --> store micro-op

Addition: svr->total += *size;

1 mov (%esi), %ebx //load micro-op
2 mov (%edi), %eax //load micro-op
3 add %ebx, %eax //addition
4 mov %eax, (%edi) //store micro-op

Load: *size = *(svr -> cur_max);

1 mov (%esi), %ebx //load micro-op
2 mov (%edi), %eax //load micro-op
3 mov 0xb(%ebx), %eax //load
4 mov %eax, (%edx) //store micro-op
Data-Oriented Gadgets

• x86 instruction sequence
  – show in normal execution (CFI)
  – save results in memory
  – load micro-op --> semantics
    micro-op --> store micro-op

Addition: \texttt{srv->total += \*size;}

1. \texttt{mov (%esi), %ebx} //load micro-op
2. \texttt{mov (%edi), %eax} //load micro-op
3. \texttt{add %ebx, %eax} //addition
4. \texttt{mov %eax, (%edi)} //store micro-op

Load: \texttt{*size = *(srv ->cur_max);}

1. \texttt{mov (%esi), %ebx} //load micro-op
2. \texttt{mov (%edi), %eax} //load micro-op
3. \texttt{mov 0xb(%ebx), %eax} //load
4. \texttt{mov %eax, (%edx)} //store micro-op
Gadget Dispatcher

corruptible by mem-err

loop → selector

round1
round2
round3
......
roundN
Gadget Dispatcher

- Chain data-oriented gadgets “legitimately”
  - **loop** ---> repeatedly invoke gadgets
  - **selector** ---> selectively activate gadgets
Chain data-oriented gadgets "legitimately"

- loop ---> repeatedly invoke gadgets
- selector ---> selectively activate gadgets
- Chain data-oriented gadgets "legitimately"
  - **loop** --> repeatedly invoke gadgets
  - **selector** --> selectively activate gadgets
Gadget Dispatcher

- Chain data-oriented gadgets "legitimately"
  - loop ---> repeatedly invoke gadgets
  - selector ---> selectively activate gadgets
• Chain data-oriented gadgets "legitimately"
  – loop ---> repeatedly invoke gadgets
  – selector ---> selectively activate gadgets
Gadget Dispatcher

- Chain data-oriented gadgets "legitimately"
  - `loop` ---> repeatedly invoke gadgets
  - `selector` ---> selectively activate gadgets

```c
while (quota--) {
    // loop
    readData(sockfd, buf);   // selector
    if(*type == NONE ) break;
    if(*type == STREAM) *size = *(srv->cur_max);
    else{ srv->typ = *type; srv->total += *size; }
}
```
Turing-completeness

• DOP emulates a minimal language *MINDOP*
  – *MINDOP* is Turing-complete

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Statements In C</th>
<th>Data-Oriented Gadgets in DOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic / logical</td>
<td>a op b</td>
<td>*p op *q</td>
</tr>
<tr>
<td>assignment</td>
<td>a = b</td>
<td>*p = *q</td>
</tr>
<tr>
<td>load</td>
<td>a = *b</td>
<td>*p = **q</td>
</tr>
<tr>
<td>store</td>
<td>*a = b</td>
<td>**p = *q</td>
</tr>
<tr>
<td>jump</td>
<td>goto L</td>
<td>vpc = &amp;input</td>
</tr>
<tr>
<td>conditional jump</td>
<td>if (a) goto L</td>
<td>vpc = &amp;input if *p</td>
</tr>
</tbody>
</table>

p – &a;    q – &b;    op – any arithmetic / logical operation
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    } //...(code skipped)...
}
• Gadget identification
  – statically identify load-semantics-store chain from LLVM IR
• Gadget identification
  – statically identify load-semantics-store chain from LLVM IR

• Dispatcher identification
  – static identify loops with gadgets from LLVM IR
Attack Construction

• Gadget identification
  – statically identify load-semantics-store chain from LLVM IR

• Dispatcher identification
  – static identify loops with gadgets from LLVM IR

• Gadget stitching
  – select gadgets and dispatchers (manual)
  – check stitchability (manual)
Evaluation
Evaluation – Feasibility

9 x86 programs with 9 vulnerabilities

- Nginx, ProFTPD, Wu-FTPD, sshd, Bitcoind,
- Wireshark, sudo, musl libc, mcrypt
Evaluation – Feasibility

9 x86 programs with 9 vulnerabilities
- Nginx, ProFTPD, Wu-FTPD, sshd, Bitcoind,
- Wireshark, sudo, musl libc, mcrypt

• x86 Gadgets
  - 7518 in total, 1273 reachable via selected CVEs
  - 8 programs can simulate all MINDOP operations

• x86 Dispatchers
  - 1443 in total, 110 reachable from selected CVEs
Evaluation – Feasibility

9 x86 programs with 9 vulnerabilities
- Nginx, ProFTPD, Wu-FTPD, sshd, Bitcoind,
- Wireshark, sudo, musl libc, mcrypt

• x86 Gadgets
  - 7518 in total, 1273 reachable via selected CVEs
  - 8 programs can simulate all MINDOP operations

• x86 Dispatchers
  - 1443 in total, 110 reachable from selected CVEs

• 2 programs can build Turing-complete attack
• 3 end-to-end attacks
Case Study: Bypassing Randomization

• Previous methods
  – information leakage to network

• Defeat ASLR w/o address leakage to network?
Case Study: Bypassing Randomization

• Previous methods
  – information leakage to network
• Defeat ASLR w/o address leakage to network?
• Vulnerable *ProFTPD*
  – use OpenSSL for authentication
  – a dereference chain to the private key
Case Study: Bypassing Randomization

• Previous methods
  – information leakage to network

• Defeat ASLR w/o address leakage to network?

• Vulnerable ProFTPD
  – use OpenSSL for authentication
  – a dereference chain to the private key

```c
@0x080dbc28
SSL_CTX * ssl_ctx

struct cert_st * cert

struct rsa_st * rsa

BN_ULONG * d2

BIGNUM * d1

CERT_PKEY * key

EVP_PKEY*privatekey
```
Case Study: Bypassing Randomization

• Gadgets

<table>
<thead>
<tr>
<th></th>
<th>MOV</th>
<th>ADD</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*p = *q</td>
<td>*x = *x + offset</td>
<td>*Z = **Y</td>
</tr>
</tbody>
</table>
Case Study: Bypassing Randomization

- Gadgets

<table>
<thead>
<tr>
<th>MOV</th>
<th>*p = *q</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>*X = *X + offset</td>
</tr>
<tr>
<td>LOAD</td>
<td>*Z = **Y</td>
</tr>
</tbody>
</table>

- Dispatcher

```c
while (1) {
    user_request = get_user_request();
    dispatch(user_request);
    func1() { memory_error; MOV; }
    func2() { ADD; }
    func3() { LOAD; }
}
```

![Diagram showing the relationship between variables and functions]

[@0x080dbc28]

- SSL_CTX * ssl_ctx
- BN_UULONG * d2
- BIGNUM * d1
- struct cert_st * cert
- Private Key
- CERT_PKEY * key
- EVP_PKEY * privatekey
- struct rsa_st * rsa
Case Study: Bypassing Randomization

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>*X = *0x080dbc28 (ssl_ctx)</td>
</tr>
<tr>
<td>ADD</td>
<td>*X = *X + offset1</td>
</tr>
<tr>
<td>MOV</td>
<td>*Y = *X</td>
</tr>
<tr>
<td>LOAD</td>
<td>*Z = **Y</td>
</tr>
<tr>
<td>MOV</td>
<td>*0x080dbc28 = *Z (cert)</td>
</tr>
</tbody>
</table>

@0x080dbc28

- SSL_CTX * ssl_ctx
- BN_ULONG * d2
- BIGNUM * d1
- struct cert_st * cert
- CERT_PKEY * key
- EVP_PKEY*privatekey
- struct rsa_st * rsa
- Private Key
### Case Study: Bypassing Randomization

<table>
<thead>
<tr>
<th>MOV</th>
<th>*X = *0x080dbc28 (ssl_ctx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>*X = *X + offset1</td>
</tr>
<tr>
<td>MOV</td>
<td>*Y = *X</td>
</tr>
<tr>
<td>LOAD</td>
<td>*Z = **Y</td>
</tr>
<tr>
<td>MOV</td>
<td>*0x080dbc28 = *Z (cert)</td>
</tr>
</tbody>
</table>

![Diagram of variables and memory offsets]

- `ssl_ctx` at `0x080dbc28`
- `cert` at `0x080dbc28`
- `BN_ULONG d2`
- `BIGNUM d1`
- `CERT_PKEY key`
- `EVP_PKEY privatekey`
- `struct rsa_st * rsa`
- `struct cert_st * cert`
### Case Study: Bypassing Randomization

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>*X = *0x080dbc28 (ssl_ctx)</td>
</tr>
<tr>
<td>ADD</td>
<td>*X = *X + offset1</td>
</tr>
<tr>
<td>MOV</td>
<td>*Y = *X</td>
</tr>
<tr>
<td>LOAD</td>
<td>*Z = **Y</td>
</tr>
<tr>
<td>MOV</td>
<td>*0x080dbc28 = *Z (cert)</td>
</tr>
</tbody>
</table>

![Diagram showing data structures and variables](image)

- SSL_CTX * ssl_ctx
- BN_ULONG * d2
- BIGNUM * d1
- struct cert_st * cert
- CERT_PKEY * key
- EVP_PKEY*privatekey
- struct rsa_st * rsa

@0x080dbc28

- d2

Case Study: Bypassing Randomization

| MOV   | *X = *0x080dbc28 (ssl_ctx) |
| ADD   | *X = *X + offset1          |
| MOV   | *Y = *X                    |
| LOAD  | *Z = **Y                   |
| MOV   | *0x080dbc28 = *Z (cert)    |

```c
write(outsock, buf, strlen(buf));
```

@0x080dbc28

<table>
<thead>
<tr>
<th>SSL_CTX * ssl_ctx</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct cert_st * cert</td>
</tr>
<tr>
<td>BNULONG * d2</td>
</tr>
<tr>
<td>BIGNUM * d1</td>
</tr>
<tr>
<td>struct rsa_st * rsa</td>
</tr>
<tr>
<td>CERT_PKEY * key</td>
</tr>
<tr>
<td>EVP_PKEY*privatekey</td>
</tr>
</tbody>
</table>
Case Study: Bypassing Randomization

<table>
<thead>
<tr>
<th>MOV</th>
<th>*X = *0x080dbc28 (ssl_ctx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>*X = *X + offset1</td>
</tr>
<tr>
<td>MOV</td>
<td>*Y = *X</td>
</tr>
<tr>
<td>LOAD</td>
<td>*Z = **Y</td>
</tr>
<tr>
<td>MOV</td>
<td>*0x080dbc28 = *Z (cert)</td>
</tr>
</tbody>
</table>

```
write(outsock, buf, strlen(buf));
```

```
@0x080dbc28
```

```
d2
```

```
buf
```

```
@0x080dbc28
SSL_CTX * ssl_ctx
```

```
struct cert_st * cert
```

```
BN_ULONG * d2
```

```
BIGNUM * d1
```

```
Private Key
```

```
CERT_PKEY * key
```

```
EVP_PKEY*privatekey
```

```
struct rsa_st * rsa
```
Case Study: Bypassing Randomization

Private Key

BN_ULONG * d2
BIGNUM * d1
struct rsa_st * rsa

EVP_PKEY*privatekey
CERT_PKEY * key
struct cert_st * cert

@0x080dbc28
SSL_CTX * ssl_ctx

MOV *X = *0x080dbc28 (ssl_ctx)
ADD *X = *X + offset1
MOV *Y = *X
LOAD *Z = **Y
MOV *0x080dbc28 = *Z (cert)

write(outsock, buf, strlen(buf));

leak private key to network

@0x080dbc28
d2
buf
dlopen() – Dynamic Linking Interface

• Load the dynamic library into memory space
  – resolve symbols based on binary metadata
  – patch program due to relocation
  – like LoadLibrary() on Windows
**dlopen() – Dynamic Linking Interface**

• Load the dynamic library into memory space
  – resolve symbols based on binary metadata
  – patch program due to relocation
  – like `LoadLibrary()` on Windows

• Dynamic loader can do arbitrary computation*

**dlopen() – Dynamic Linking Interface**

- Load the dynamic library into memory space
  - resolve symbols based on binary metadata
  - patch program due to relocation
  - like `LoadLibrary()` on Windows

- Dynamic loader can do arbitrary computation*

The same to `dlopen()`

---

Case Study: Simulating A Network Bot

• Attacks with `dlopen`

dlopen() { head }

dynamic library list `link_map`

......
Case Study: Simulating A Network Bot

- Attacks with *dlopen*
  - send malicious payload

```
dlopen() {              }
head ……
```

dynamic library list *link_map*

```
ProFTPD's memory
```

```
Malicious payload
```

83
Case Study: Simulating A Network Bot

• Attacks with `dlopen`
  – send malicious payload
  – corrupt link list & call `dlopen`

```c
int dlopen() {
    head
    …
}
```

dynamic library list `link_map`

ProFTPD’s memory

Malicious payload
Case Study: Simulating A Network Bot

• Attacks with `dlopen`
  – send malicious payload
  – corrupt link list & call `dlopen`
Case Study: Simulating A Network Bot

- Attacks with `dlopen`
  - send malicious payload
  - corrupt link list & call `dlopen`

Malicious payload

ProFTPD’s memory

Dynamic library list `link_map`
Case Study: Simulating A Network Bot

• DOP attack addresses the problems

  – send malicious payload  invalid input
  – corrupt link list & call \texttt{dlopen}  no call to \texttt{dlopen}

\begin{tikzpicture}
  \node[draw] (head) at (0,0) {\texttt{dlopen()}};
  \node[draw] (link_map) at (2,0) {\texttt{link\_map}};
  \node[draw] (malicious) at (-2,-4) {Malicious payload};
  \node[draw] (memory) at (-2,-3) {ProFTPD's memory};

  \draw[->,red] (head) -- (malicious);
  \draw[->,blue] (link_map) -- (malicious);
  \draw[->,blue] (memory) -- (link_map);
\end{tikzpicture}
Case Study: Simulating A Network Bot

- DOP attack addresses the problems
  - construct payload in memory
  - corrupt link list & call `dlopen`

`dlopen()` { head }  

ProFTPD’s memory

Malicious payload

(1) Payload prepare
  MOV
  MOV

Dynamic library list `link_map`

invalid input  
no call to `dlopen`
Case Study: Simulating A Network Bot

- DOP attack addresses the problems
  - construct payload in memory
  - corrupt link list & call `dlopen`

Invalid input: no call to `dlopen`

Dynamic library list `link_map`
Case Study: Simulating A Network Bot

- DOP attack addresses the problems
  - construct payload in memory
  - force call to `dlopen`

```c
if (flag) {
    dlopen() { head }
}
```

Invalid input: no call to `dlopen`

Diagram:

- `ProFTPD's memory`
- `Malicious payload`
- `Dynamic library list `link_map``
- (1) Payload prepare
  - MOV
  - MOV
Case Study: Simulating A Network Bot

- DOP attack addresses the problems
  - construct payload in memory  invalid input
  - force call to `dlopen`  no call to `dlopen`

```c
if (flag) {
    dlopen() {
        head
    }
}
```

ProFTPD’s memory

Malicious payload

Dynamic library list `link_map`

(1) Payload prepare
    MOV

(2) Trigger
    MOV
    STORE
Case Study: Simulating A Network Bot

• DOP attack addresses the problems
  – construct payload in memory invalid input
  – force call to `dlopen` no call to `dlopen`

```c
if (flag) {
    dlopen() { head }
}
```

(1) Payload prepare
    MOV
    MOV

(2) Trigger
    MOV
    STORE

ProFTPD’s memory

Malicious payload

> 700 requests
Case Study: Altering Memory Permissions

• Defenses based on memory permissions
  – DEP: non-writable code
  – CFI: non-writable jump tags
Case Study: Altering Memory Permissions

• Defenses based on memory permissions
  – DEP: non-writable code
  – CFI: non-writable jump tags

• \texttt{dlopen()}: relocation
  – change any page permission to writable
  – update page content
  – change the permission back
Case Study: Altering Memory Permissions

• Defenses based on memory permissions
  – DEP: non-writable code
  – CFI: non-writable jump tags

• `dlopen()`: relocation
  – change any page permission to writable
  – update page content
  – change the permission back

• DOP attacks
  – `dlopen(code_addr, shellcode)`
Case Study: Altering Memory Permissions

• Defenses based on memory permissions
  – DEP: non-writable code
  – CFI: non-writable jump tags

• `dlopen()`: relocation
  – change any page permission to writable
  – update page content
  – change the permission back

• DOP attacks
  – `dlopen(code_addr, shellcode)`

• Code injection is back!
## Related Work

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Turing Complete?</th>
<th>Preserve CFI?</th>
<th>Independent of specific data / funcs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-control Data Attacks (Chen et al. 2005)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>COOP (Schuster et al. 2015)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FlowStitch (Hu et al. 2015)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Printf-Oriented Programming (Carlini et al. 2015)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Control Jujustu (Evans et al. 2015)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Data-Oriented Programming</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Potential Defenses

• Memory Safety
  – e.g., Cyclone (Jim et al. 2002), CCured (Necula et al. 2002), SoftBounds+CETS (Nagarakatte et al. 2009, 2010)
  – high performance overhead (> 100%)

• Data-flow Integrity
  – e.g, DFI (Castro et al. 2006), kernel DFI (Song et al. 2016)

• Fined-grained randomization in data space
  – e.g., DSR (Bhatkar et al. 2008)

• Hardware & software fault isolation
  – e.g., HDFI (Song et al. 2016), MPX
Potential Defenses

• Memory Safety
  – e.g., Cyclone (Jim et al. 2002), CCured (Necula et al. 2002), SoftBounds+CETS (Nagarakatte et al. 2009, 2010)
  – high performance overhead (> 100%)

• Data-flow Integrity
  – e.g., DFI (Castro et al. 2006), kernel DFI (Song et al. 2016)

• Fined-grained randomization in data space
  – e.g., DSR (Bhatkar et al. 2008)

• Hardware & software fault isolation
  – e.g., HDFI (Song et al. 2016), MPX

No practical defenses yet!
Conclusion

• Non-control data attacks can be Turing-complete
Conclusion

• Non-control data attacks can be Turing-complete

• Data-Oriented Programming (DOP)
  – build expressive non-control data attacks
  – independent of specific data / functions
Conclusion

• Non-control data attacks can be Turing-complete

• Data-Oriented Programming (DOP)
  – build expressive non-control data attacks
  – independent of specific data / functions

• In real-world programs, DOP can build attacks
  – bypass ASLR w/o address leakage
  – simulate a network bot
  – enable code injection
Thanks!

Hong Hu

huhong@comp.nus.edu.sg

http://www.comp.nus.edu.sg/~huhong

Non-control data attacks are available

http://huhong-nus.github.io/advanced-DOP/