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

CFG

Memory space

Data

Code
Control Attacks are Getting Harder

• Code injection
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  \[\leftrightarrow\]  Control Flow Integrity

![CFG w/ CFI](image-url)
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
```

```c
//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

---

+ 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, *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

```
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 }
```
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]

```c
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--) {
   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

CFG w/ CFI

Malicious Computation

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 { /* server struct */
    int *cur_max, total, typ;
} *srv;

int quota = MAXCONN; /* quota int */
int *size, *type;
char buf[MAXLEN]; /* buffer */

size = &buf[8]; /* initialize size */
type = &buf[12]; /* initialize type */
...

while (quota--) {
    readData(sockfd, buf); /* read data */
    if (*type == NONE) break; /* check type */
    if (*type == STREAM) {
        *size = srv->cur_max; /* set size */
    } else {
        srv->typ = *type; /* set typ */
        srv->total += *size; /* add size to total */
    }
} /*...following code skipped...*/
```

**Motivating Example**

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

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

vulnerable program

malicious computation

simulate ?
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  }
```

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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Motivating Example (cont.)

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

Motivating Example (cont.)

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 }

malicious computation

simulate ?

vulnerable program

Memory space

heap

stack

buf[] type size quota srv

addend

list

next prop

next prop

cur_max total typ
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;
    }
}
```

Tasks

- simulate

Vulnerable program

![Memory space diagram]

malicious computation

heap

stack

buf[]

type

size

quota

srv

addend
cur_max

total

typ

list

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;
    }
}
```

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

Motivating Example (cont.)

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

\begin{tikzpicture}[node distance=2cm]
  \node (list) {list};
  \node [above of=list] (addend) {addend};
  \node [left of=list] (buf) {buf[]};
  \node [right of=list] (type) {type};
  \node [right of=type] (size) {size};
  \node [right of=size] (quota) {quota};
  \node [right of=quota] (srv) {srv};
  \node [below of=list] (next) {next};
  \node [right of=next] (prop) {prop};

  \draw [->] (list) -- (addend);
  \draw [->] (uid) -- (buf);
  \draw [->] (addend) -- (type);
  \draw [->] (type) -- (size);
  \draw [->] (size) -- (quota);
  \draw [->] (quota) -- (srv);
  \draw [->] (srv) -- (next);
  \draw [->] (next) -- (prop);
  \draw [->] (buf) -- (uid);
  \draw [->] (uid) -- (type);
  \draw [->] (type) -- (size);
  \draw [->] (size) -- (quota);
  \draw [->] (quota) -- (srv);
  \draw [->] (srv) -- (next);
  \draw [->] (next) -- (prop);

  \node [above right of=srv] (cur_max) {cur_max};
  \node [right of=cur_max] (total) {total};
  \node [right of=total] (typ) {typ};

  \node [below right of=list] (heap) {heap};
  \node [above right of=list] (stack) {stack};
\end{tikzpicture}
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;
}
```

Vulnerable program

Malicious computation

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

Motivating Example (cont.)

while (list != NULL; list = list->next) {
    list->prop += addend;
    cur_max
    total
typ

malicious computation

Memory space

heap

stack

vulnerable program
Motivating Example (cont.)

Memory space

```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
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;
```
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 ?

heap

stack

Memory space

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

```c
addend
```

```c
list
```

```c
cur_max
```

```c
total
```

```c
typ
```

```c
next
```

```c
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;
```

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>
<tbody>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addend</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cur_max</td>
<td>total</td>
<td>typ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>prop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td>prop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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;
```

Memory space

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

```
malicious computation
simulate ?
```

heap

```
list
```

```
addend
```

```
cur_max
```
```
total
```
```
typ
```

stack

```
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

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

Malicious computation

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 ?

---

Memory space

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

Stack

```
cur_max total typ
```

Heap

```
STREAM
```

```
addend
```

```
list
```

```
next prop
```

```
next prop
```
Motivating Example (cont.)

```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 }
```

malicious computation

simulate ?

heap

```
Memory space

buf[]  type  size  quota  srv

stack

cur_max  total  typ

next  prop

STREAM

addend

list

next  prop
```

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;
    }
}
```

malicious computation

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

vulnerable program

---

Memory space

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

```
STREAM
```

```text
heap
```

```text
list
```

```text
next  prop
```

```
cur_max  total  typ
```

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;
    }
}
```

The code contains a vulnerable program that can be exploited by a malicious computation. The `readData` function reads data into the buffer `buf`, and the `if` statements check the type of the data. If the type is `NONE`, the loop breaks. If the type is `STREAM`, the size is calculated from the current maximum size. Otherwise, the type is stored, and the total size is increased.

The memory space diagram visualizes the data structures and connections in the program. The `buf` array stores data, while the `type`, `size`, and `quota` variables are used to track the type and size information. The `srv` pointer points to the server structure, and the `cur_max`, `total`, and `typ` variables are used to store current maximum, total, and type information, respectively. The `heap` and `stack` segments represent the memory allocation areas, with arrows indicating data movement and access.
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;
```

Memory space

```
buf[]  type  size  quota  srv

STREAM

list          addend

next  prop

next  prop

cur_max  total  typ

heap

stack

malicious computation

simulate ✓
```

vulnerable program
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;`

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>mov (%esi), %ebx</code></td>
<td>//load micro-op</td>
</tr>
<tr>
<td>2</td>
<td><code>mov (%edi), %eax</code></td>
<td>//load micro-op</td>
</tr>
<tr>
<td>3</td>
<td><code>add %ebx, %eax</code></td>
<td>//addition</td>
</tr>
<tr>
<td>4</td>
<td><code>mov %eax, (%edi)</code></td>
<td>//store micro-op</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Addition: srv-&gt;total += *size;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  mov (%esi), %ebx          //load micro-op</td>
</tr>
<tr>
<td>2  mov (%edi), %eax          //load micro-op</td>
</tr>
<tr>
<td>3  add %ebx, %eax            //addition</td>
</tr>
<tr>
<td>4  mov %eax, (%edi)          //store micro-op</td>
</tr>
</tbody>
</table>

CFG

Memory space
Data-Oriented Gadgets

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

---

**Load:**
```
*size = *(srv ->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
  – \textit{load} micro-op \textit{--> semantics}

\textit{micro-op} \textit{--> store} micro-op

\begin{center}
\begin{tabular}{l}
\textbf{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 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{l}
\textbf{Load: } *size = *(srv ->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 \\
\end{tabular}
\end{center}
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** --- repeatently invoke gadgets
  - **selector** --- selectively activate gadgets

Diagram:
- Loop to Selector
- Repeatedly invoke gadgets in rounds (round1, round2, round3, round4, ... roundN)

Corruptible by mem-err
• 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

```java
while (quota--) {
    readData(sockfd, buf);  // loop
    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>
<tr>
<td>p – &amp;a; q – &amp;b; op – any arithmetic / logical operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)...
}
Attack Construction

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    }  //...(code skipped)...
15  }

• Gadget identification
  – statically identify load-semantics-store chain from LLVM IR
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)...
} //...(code skipped)...

• 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)

```c
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)...
}
```
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

@0x080dbc28

SSL_CTX * ssl_ctx

struct cert_st * cert

BN_UULONG * d2

BIGNUM * d1

struct rsa_st * rsa

CERT_PKEY * key

EVP_PKEY*privatekey

Private Key
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>

@0x080dbc28

SSL_CTX * ssl_ctx

struct cert_st * cert

CERT_PKEY * key

Private Key

BN_ULONG * d2

BIGNUM * d1

struct rsa_st * rsa

EVP_PKEY*privatekey
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

```
while (1) {
    user_request =
    get_user_request();
    dispatch(user_request);
}
```

```
func1() { memory_error; MOV;}
func2() { ADD; }
func3() { LOAD; }
```
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>
Case Study: Bypassing Randomization

### MOV
*X = *0x080dbc28 (ssl_ctx)

### ADD
*X = *X + offset1

### MOV
*Y = *X

### LOAD
*Z = **Y

### MOV
*0x080dbc28 = *Z (cert)

---

![Diagram](image_url)

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

@0x080dbc28

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

@0x080dbc28

- d2

MOV

ADD

MOV

LOAD

MOV
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

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

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

@0x080dbc28

movd dqssl_ctx
add dqssl_ctx,offset1
mov dqssl_ctx
leaq dqssl_ctx(dqssl_ctx)

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

---

<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>
Case Study: Bypassing Randomization

Leak private key to network

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

<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>

@0x080dbc28

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

---

Case Study: Simulating A Network Bot

- Attacks with `dlopen`

```c
void dlopen() {
    head
}
```

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

• Attacks with `dlopen`
  – send malicious payload

```
dlopen() {              }
```

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`

```
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`

```c
void dlopen() {
    struct link_head { head }
    // dynamic library list link_map
}
```

ProFTPD’s memory

Malicious payload

Invalid input
Case Study: Simulating A Network Bot

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

• DOP attack addresses the problems
  – send malicious payload
  – corrupt link list & call `dlopen`

• DOP attack addresses the problems
  – 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`

`dlopen()` allows arbitrary computation
- send malicious payload
- corrupt link list & call `dlopen`

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

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

ProFTPD’s memory

Malicious payload

(1) Payload prepare

MOV

MOV

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

- DOP attack addresses the problems
  - construct payload in memory  invalid input
  - corrupt link list & call `dlopen`  no call to `dlopen`

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

ProFTPD’s memory

Malicious payload

(1) Payload prepare
    MOV
    MOV

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`

Dynamic library list `link_map`

1. Malicious payload
   - `MOV` instructions

ProFTPD’s memory

(1) Payload
   - prepare
   - `MOV`
Case Study: Simulating A Network Bot

• DOP attack addresses the problems
  – construct payload in memory
  – force call to \texttt{dlopen}

if (flag) {
    \texttt{dlopen()} \{ head \}
}

dynamic library list \texttt{link\_map}

(2) Trigger
MOV
STORE

(1) Payload
prepare
MOV
MOV

ProFTPD's memory

Malicious payload
Case Study: Simulating A Network Bot

- DOP attack addresses the problems
  - construct payload in memory
  - force call to \textit{dlopen}

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

Dynamic library list \textit{link\_map}

- \textbf{(2) Trigger}
  - MOV
  - STORE

ProFTPD's memory

- \textbf{(1) Payload}
  - prepare
  - MOV
  - MOV

> 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

• `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><strong>Data-Oriented Programming</strong></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/