“Weird Machines” in ELF: A Spotlight on the Underappreciated Metadata

Rebecca “bx” Shapiro
bx@cs.dartmouth.edu

Sergey Bratus
sergey@cs.dartmouth.edu

Sean W. Smith
sws@cs.dartmouth.edu

Dartmouth College

WOOT Aug 13, 2013
Overview

• History of metadata
  – In exploitation
  – In defense
• Motivation: why ELF metadata
• Overview of runtime loading
• Cobbler: ELF metadata-driven computation
• Conclusion
Our contributions

- Highlight metadata as interesting attack vector
- Built Turing-complete computation environment
  - ELF metadata $\rightarrow$ instructions
  - Runtime loader $\rightarrow$ machine
- Highlight loader's role in composition & trust
One of these things is not like the others...

- Trojans/viruses
- SQL injection
- Cross site scripting
- Stack smashing (Aleph One)
- Return-oriented programming
One of these things is not like the others...

- Trojans/viruses
- SQL injection
- Cross site scripting
- Stack smashing (Aleph One)
- Return-oriented programming

bring your own code
One of these things is not like the others...

- Trojans/viruses
- SQL injection
- Cross site scripting
- Stack smashing (Aleph One)
- Return-oriented programming

bring your own code

bring your own data
BYO “Code” Attacks

• Idea
  – Untrusted-code injection == bad

• Defenses
  – Antivirus
  – Data Execution Prevention (DEP)
  – Address Space Layout Randomization (ASLR)
  – Code signing
  – Input sanitation
Malicious data “execution”

• Idea
  – Data → virtual machine bytecode
  – Execution environment → virtual machine
  – “Weird machine”

• Example – ROP
  – Injected well-formed stack frames → bytecode
  – Processor → virtual machine

• Defenses
  – ASLR (for ROP)
  – Code signing (sometimes)
  – Input sanity checking (sometimes)
Role of data in attacks

• Typically straightforward: overwrite function pointers
  – Stack smashing
  – Heap smashing
  – Viruses
  – (Means to an end)

• But not always
  – LOCREATE (Scapy)
    • PE metadata-driven unpacker
  – Signed PE code injection (Glücksmann)
    • Took advantage unsigned signature metadata
ELF Metadata-driven “weird machines”

- Most defenses focus on BYO code attacks
  - DEP, Antivirus
- Not deterred by existing ASLR
  - ELF metadata “knows” address layout
- Code never changed
- Well-formed metadata
  - Doesn't fail parsing checks
- Metadata more **trusted** than code
  - Defines address space layout
  - Not focus of antivirus
  - Achilles' heal of codesigning
Data-driven weird Turing machines

- Stack frames drive ROP machines
- DWARF error handling metadata (Oakley, Bratus)
- HTML + CSS3 (Fox-Epstein)
- C++ Templates (Veldhuizen)
Loading, Linking, and ELF
How to execute an ELF

exec("hello");

exec("hello")
How to execute an ELF

exec("hello");

exec("hello")  ➔  kernel
How to execute an ELF

exec("hello");
How to execute an ELF

exec("hello");
How to execute an ELF

exec("hello");
How to execute an ELF

exec("hello");

exec("hello") ➔ kernel ➔ RTLD_START() ➔ ld.so ➔ start() ➔ hello

hello  ld.so

libc.so
How to execute an ELF

exec("hello");
How to execute an ELF

```
exec("hello");
```

Runtime loading
How to execute an ELF

```
exec("hello");
```
**ELF metadata in a nutshell**

**ELF executable**
- Section headers
- Segment headers
- Dynamic table
- Symbol metadata
- Relocation metadata
- Other ELF metadata
- Program data
- Program code (text)

what ELF contains
- expected memory map
- ELF metadata summary for RTLD
- in/exported functions/objects
- virtual addrs to patch

Symbol + relocation metadata = virtual addr patching instructions
Introducing: Cobbler
Our toolkit for taming the ELF weird machine

Warning: the following is specific to Ubuntu 11.10's eglibc-2.13 on amd64
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
Cobbler: BrainF***-to-ELF compiler

- BF: an esoteric Turing-complete language
- If BF program finishes → executable cleanly runs
BF-enhanced ELF executable

- Section headers
- Segment headers
- Dynamic table
- Symbol metadata
- Relocation metadata
- Other ELF metadata
- Program data
- Program code (text)
- BF Symbol metadata
- BF Relocation metadata
The BF-ELF virtual machine

exec("hello") → kernel

RTLD_START() → ld.so

start() → hello

_dl_fixup()
Executing BF-ELF

... in the runtime loader

- Required libraries loaded
- link_map structures created
- Relocations performed
Cobbler primitives
add, mov, jnz

• **Symbol metadata** act as registers
  – Symbol value → register contents
  – Memory mapped
  – Contains metadata

• **Bytecode** built from relocation metadata
Cobbler bytecode

- Types of operands
  - **Immediate** – value in relocation entry ($0x01)
  - **Direct** – address of value in relocation entry (*0xdeadbeef)
  - **Register** – value in “register” (%reg)
    - Register = symbol specified by relocation entry
  - **Register indirect** – register contains address of the value ([%reg])
- All **destinations** are specified in **direct** mode
Mov (immediate)

- mov <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = immediate

Example: mov *0xbeef0000, $0x04
Mov (immediate)

- mov <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = immediate

Example: mov *0xbeef0000, $0x04

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_RELATIVE</td>
<td>beef0008</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>

0x00068000
0xbeef0000
0xbeef0008
Mov (immediate)

- mov <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = immediate

Example: mov *0xbeef0000, $0x04

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_RELATIVE</td>
<td>beef0008</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>
Mov (immediate)

- mov <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = immediate

Example: mov *0xbeef0000, $0x04

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_RELATIVE</td>
<td>beef0008</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>

---

0x00068000
0xbeef0000
0xbeef0008
Mov (immediate)

- `mov <destination>, <value>`
  - `<destination>` = direct (address of destination)
  - `<value>` = immediate

**Example:** `mov *0xbeef0000, $0x04`

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_RELATIVE</td>
<td>beef0008</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>

0x00068000
0xbeef0000
0xbeef0008

0x0000000000000000
0x0000000000000003
Mov (immediate)

- mov <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = immediate

Example: mov *0xbeef0000, $0x04

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_RELATIVE</td>
<td>beef0008</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>
Mov (indirect)

Example: mov *0xbeef0000, [%foo]

```
<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>0xbeef0000</td>
<td>FUNC</td>
<td>X</td>
<td>8</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_COPY</td>
<td>0xbeef0000</td>
<td>foo</td>
<td>X</td>
</tr>
</tbody>
</table>

0x00064000

0x00068000

0x00068000

0x0000000000000000

0x0000000000000002

(See publication)
Addition

- add <destination>, <addend 1>, <addend 2>
  - <destination> = direct (address of destination)
  - <addend 1> = register
  - <addend 2> = immediate
Addition

Example: add *0xbeef0000, %add1, $0x02

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td>0x1</td>
<td>FUNC</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>beef0000</td>
<td>add1</td>
<td>2</td>
</tr>
</tbody>
</table>

Example: add *0xbeef0000, %add1, $0x02
Addition

Example: add *0xbeef0000, %add1, $0x02
Addition

Example: add *0xbeef0000, %add1, $0x02

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td>0x1</td>
<td>FUNC</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>beef0000</td>
<td>add1</td>
<td>2</td>
</tr>
</tbody>
</table>

0x00064000

0x00068000

0xbeef0000

0xbeef0008
Addition

Example: add *0xbeef0000, %add1, $0x02
Addition

Example: add *0xbeef0000, %add1, $0x02
Jump if not zero

- jnz <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = direct (address of value to check)
One does not simply tell the loader to *not* process the next relocation entry...
One does not simply tell the loader to *not* process the next relocation entry...

(jnz cannot be implemented with a single relocation entry)
How relocation entries are processed
(pseudocode)

\[
\text{while} \ (lm != \text{NULL}) \ \{ \\
\text{r} = lm->\text{dyn}[\text{DT\_RELA}]; \\
\text{end} = r + lm->\text{dyn}[\text{DT\_RELASZ}]; \\
\text{for} \ (r; \ r < \text{end}; \ r++ \ ) \ \{ \\
\text{relocate}(lm, \ r, \ &\text{dyn}[\text{DT\_SYM}]); \\
\} \\
\text{lm} = \text{lm}\rightarrow\text{prev}; \\
\} \\
\]
Clobbering ld.so data

1. Preparation:
   - lm->prev = lm
   - lm->dyn[RELA]
     - Next rela to process
   - lm->dyn[RELASZ]
     - # to process

2. Stop execution:
   - end = 0 to break out of loop

```c
while (lm != NULL) {
    r = lm->dyn[RELA];
    end = r + lm->dyn[RELASZ];
    for (r; r < end; r++) {
        relocate(lm, r, &dyn[SYM]);
    }
    lm = lm->prev;
}
```
Clobbering ld.so data

1. Preparation:
   - lm->prev = lm
   - lm->dyn[RELA]
     - Next rela to process
   - lm->dyn[RELASZ]
     - # to process

2. Stop execution:
   - end = 0 to break out of loop

while (lm != NULL) {
    r = lm->dyn[RELA];
    end = r + lm->dyn[RELASZ];
    for (r; r < end; r++) {
        relocate(lm, r, &dyn[SYM]);
    }
    lm = lm->prev;
}
Clobbering ld.so data

1. Preparation:
   - lm->prev = lm
   - lm->dyn[RELDA]
     - Next rela to process
   - lm->dyn[RELASZ]
     - # to process

2. Stop execution:
   - end = 0 to break out of loop

```c
while (lm != NULL) {
    r = lm->dyn[RELDA];
    end = r + lm->dyn[RELASZ];
    for (r; r < end; r++) {
        relocate(lm, r, &dyn[SYM]);
    }
    lm = lm->prev;
}
```
Clobbering ld.so data

1. Preparation:
   - lm->prev = lm
   - lm->dyn[RELA]
     - Next rela to process
   - lm->dyn[RELASZ]
     - # to process

2. Stop execution:
   - end = 0 to break out of loop

```c
while (lm != NULL) {
    r = lm->dyn[RELA];
    end = r + lm->dyn[RELASZ];
    for (r; r < end; r++) {
        relocate(lm, r, &dyn[SYM]);
    }
    lm = lm->prev;
}
```
Clobbering ld.so data

1. Preparation:
   - \texttt{lm->prev = lm}
   - \texttt{lm->dyn[RELA]}
     - Next rela to process
   - \texttt{lm->dyn[RELASZ]}
     - \# to process

2. Stop execution:
   - \texttt{end = 0} to break out of loop

\begin{verbatim}
while (lm != NULL) {
    r = lm->dyn[RELA];
    end = r + lm->dyn[RELASZ];
    for (r; r < end; r++) {
        relocate(lm, r, &dyn[SYM]);
    }
    lm = lm->prev;
}
\end{verbatim}
Clobbering ld.so data

1. lm->prev = lm
   - mov *(&(lm->prev)), $(&lm))

2. lm->dyn[RELA]
   - mov *(&(lm->dyn[RELA])), $(&next_rela))

3. lm->dyn[RELASZ]
   - mov *(&(lm->dyn[RELA])), #rela*sizeof(rela))

4. end
   - mov *(&end, $0)
Conditional Branching

- Have relocation entries do necessary bookkeeping
- Use IFUNC symbol with a value that points to code that returns 0
  - IFUNC symbol value treated as function pointer
  - Symbols of type IFUNC only processed as function if st_shndx != 0
  - Move value to test to ifunc's st_shndx
    - mov *(&(ifunc_sym.st_shndx), <test value>)
- Finally ....
Jump if not zero – final step

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>?</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

```
0x0000000000000000
0x0000000000000000
0x00064000
0x00068000
0x00068030
0x00068048
0xdeadbee0
0xdeadbee8

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
```
Jump if not zero – final step

```c
for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
```
Jump if not zero – final step

```
for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
```
Jump if not zero – final step if zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}

0x00064000
0x00068000
0x00068018
0x00068030
0x00068048
0xdeadbee0
0xdeadbee8

0x00068018
0xdeadbee8

0x00068048
Jump if not zero – final step
if zero

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
Jump if not zero – final step

if zero

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

0x00064000

0x00068000
0x00068018
0x00068030
0x00068048
0xdeadbee0
0xdeadbee8

0x00068018
0x00068048

0x0000000000000000
0x0000000000068048
Jump if not zero – final step
if zero

```
for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
```
Jump if not zero – final step
if zero

for (r; r < end; r++ ) {
    relocate(lm, r, &dyn[SYM]);
}
Jump if not zero – final step
if zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

\[0x000000000000f0020\]

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}

0xf0020 > 0x68049
Jump if not zero – final step
if not zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
Jump if not zero – final step
if not zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

0x00064000

for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}

0x00068000
0x00068018
0x00068030
0x00068048

0xdeadbee0
0xdeadbee8

0x000000000000000000000000000068048
Jump if not zero – final step
if not zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SYM</td>
<td>0xdeadbee8</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

0 < 0x68049

```c
for (r; r < end; r++) {
    relocate(lm, r, &dyn[SYM]);
}
```
Challenges

• Preserve/restore ELF's existing metadata
• Sanity checks make branching harder
• Address of ld.so + it's data is randomized
  – Addr of executable's link_map at fixed virtual addr
• <end> is stored on stack
  – Can locate stack at runtime
What else can ELF metadata do?

- Locate stack (see code)
- Locate all mapped libraries (see paper)
- Redirect library calls
  - To insert root shell backdoor into ping (see paper)
- Perform function calls
  - (controlling arguments are tricky)
Conclusion

- Code injection can lead to Bad Things
- Defenders focus on code injection
- Data can be just as powerful
  - RTLD ELF relocation engine → Turing complete!
  - ELF don't care about DEP or ASLR
  - Loader implicitly trusts ELF
- Adaptability v. computing power
  - Is there a good balance?

Image source: Jaganath on Wikipedia CC BY-SA 3.0
End.

● Thanks you!
  - The Dartmouth Trust Lab
    - Sergey Bratus, Sean Smith
  - WOOT reviewers (you know who you are)
  - Qualcomm
  - Our sponsors*

*This work was sponsored in part by the DOE and Intel
Questions?

elf-bf-tools repository on github

https://github.com/bx/elf-bf-tools

@bxsays on twitter