A2C: Self Destructing Exploit Executions via Input Perturbation

Yonghwi Kwon¹, Brendan Saltaformaggio¹, I Luk Kim¹, Kyu Hyung Lee², Xiangyu Zhang¹, and Dongyan Xu¹

¹Department of Computer Science, Purdue University
²Department of Computer Science, University of Georgia
In most attacks, attackers need to inject malicious payloads and they are brittle.
Our Solution: A2C

Observation

Malicious Input: ...01010101010...

Malicious Payload: Shellcode/ROP

Shellcode (Payload)

| 31 | c0 31 f6 50 5f 50 b0 66 6a 01 5b 53 6a 02 89 e1 cd 80 96 |

Corresponding Instructions

xor eax, eax; xor esi, esi; push eax; pop edi; push eax; ...

XOR 0xAA

| 9b | 6a 9b 5c fa f5 fa 1a cc c0 ab f1 f9 c0 a8 23 4b 67 2a 3c |

fwait; push 0xffffffff9b; pop esp; cli;cmc;cli;sbbcl,ah;shr...
Our Solution: A2C
Benign execution

Input (HTTP request)
POST /index.php HTTP/1.1 ...

Encoded input
ONRS..hmcdw-ogo.GSSO.0-0 ...

Web server
Parses/Processes Inputs
Generates Outputs

Encoded input
ONRS.hmcdw-ogo.GSSO.0-0 ...

Output (HTML page)
<html><head><title>....</html>
Our Solution: A2C

Idea

Inputs

Encoded inputs

Program

Exploit

Payload is *encoded*: Attack Failed

Benign request
Our Solution: A2C
Why payloads are not decoded?

Decoding based on input processing semantics
We statically analyze a program and decode when inputs are used by the program (as intended data)

Inputs should be data, not code
A2C allows inputs to be accessed as (intended types of) data, but breaks if they are code (or unintended types of data (e.g., ROP gadgets))
Our Solution: A2C
Overview

Original Program → Program Analysis (Constraint Solving + Static Analysis) → Instrumented Program + Runtime Support
Step 1: Program Analysis

When to encode and decode?

When to encode?
Encode incoming inputs from *untrusted sources* at library calls (e.g., recv, read)

When to decode?
Decode when the encoded values are consumed by the *program’s input processing logic*
Program Analysis

When to decode?

Encoded Inputs → Program → Outputs

- Copy
- Read/Compare (Parse)
- Conversion (e.g., Charset conversion)
- Computation
Program Analysis
When to decode?

- Encoded Inputs
- Program
- Outputs

- Copy
- Read/Compare (Parse)
- Conversion (e.g., Charset conversion)
- Computation

- No Decode
- Decode

Questions:

10
Program Analysis
Can an attacker control results?

Malicious Inputs (Payload)

Conversion (e.g., Charset conversion)

Computation

Operation 1

A → Gear → B → Gear → X

Y
Program Analysis
Can an attacker control results?

Malicious Inputs (Payload)

Conversion (e.g., Charset conversion)

Computation

Operation 2

A

B

1

1

?
Program Analysis
Not Sure? Ask Constraint Solver!

// Declarations (Data Types)

unsigned int m7[...][...];
unsigned short img[...][...];
unsigned short mpr[...][...];
...

// Transformative Operations
for (int x = 0; ...; x++)
  for (int y = 0; ...; y++)
    m7[x][y] = img[...][...] - mpr[...][...];
6. \( m7[x][y] = \text{img}[][][...] - \text{mpr}[][][...]; \)

; Constraints for Operations (img - mpr)  
\( m7[0,1,2,3] = \text{img}[0,1,2,3] - \text{mpr}[0,1,2,3] \)

; Constraints for the range of unsigned short  
\( 0 \leq \text{img}[0,1,2,3] \quad \land \quad 0 \leq \text{mpr}[0,1,2,3] \)
\( \text{img}[0,1,2,3] \leq 65535 \quad \land \quad \text{mpr}[0,1,2,3] \leq 65535 \)

; Constraints for Payloads (\( n \) will select a payload)  
\( m7[0,1,2,3] = \text{payload}[n, n+1, n+2, n+3] \)
Program Analysis
Not Sure? Ask Constraint Solver!

Z3 Solver

Payloads
Program Analysis
Not Sure? Ask Constraint Solver!

Constraint Solver returns ...

SAT: Attackers can control
TIMEOUT and UNKNOWN: Don’t know ➔
Attackers might control!

UNSAT ➔ Attackers cannot control!
Decoding Frontier
Exploitable and Post-Exploitable Space

Encoded Inputs -> Program

- Copy: No Decode
- Conversion (e.g., Charset conversion): No Decode
- Simple Computation: No Decode
- Read/Compare (Parse): Decode
- Certain Complex Computation: Decode

Outputs
Decoding Frontier

Exploitable and Post-Exploitable Space

Encoded Inputs → Program → Outputs

Encoded

- Copy
- Conversion (e.g., Charset conversion)
- Simple Computation

Decoded

- Read/Compare (Parse)
- Certain Complex Computation

Exploitable Space

Decoding Frontier

Post-exploitable Space
Step 2: Instrumentation

When to encode?
- Encode incoming inputs from *untrusted sources* at library calls (e.g., recv, read)
- Encode “*constants*” that can be written to *encoded buffers* (Details in the paper)

When to decode?
- Decode when encoded values are consumed by the program’s input processing logic
- Decode *permanently* at decoding frontier
Evaluation
Performance (18 real world apps + SPEC CPU2006)

18 real world apps
Average: 6.11%

SPEC CPU2006
Average: 8.13%

Average of all (30 programs): 6.94%
Evaluation
Effectiveness

23 different exploits on 18 programs
Tested 100 payloads (50 shellcode/50 ROP) for each program

3.0
Avg. # of instructions executed in payloads
XOR with 0xAA on malicious payloads.
Only 3-4 instructions are executed and these are meaningless.

0.1
Avg. # of ROP gadgets executed
Almost no ROP gadgets were executed.

Mutation will break malicious payloads execution, and it will break early.
Discussion
Limitations

Attacks in Post-exploitable Space
We use a large pool of payload test cases that models the distribution of valid payloads to determine the DF with strong probabilistic guarantees.

Memory Disclosure
We use a different dictionary (encoding key) for each buffer and each request. Knowing a previous buffer’s dictionary does not help in subsequent attacks.
Related Works

**CFI** Practical CFI (V. van der Veen et al. in CCS’15, B. Niu et al. in CCS’15, C. Tice et al. in SEC’14, C. Zhang et al. in SP’13, M. Zhang et al. in SEC’13, V. Pappas et al. in SEC’13, Y. Xia et al. in DSN’12, ...), SafeDispatch (D. Jang et al. in NDSS’14), Control Flow Locking (T. Bletsch et al. in ACSAC’11), ...

**Malicious Payloads Detection** Z. Liang et al. in CCS’05, T. Toth et al. in RAID’02, P. Fogla et al. in SEC’06, M. Polychronakis et al. in RAID’07, K. Snow et al. in SEC’11, ....

**Randomizations** ASLR (R. Wartell et al. in CCS’12, V. Pappas et al. in SP’12, D. Bigelow et al. in CCS’15, S. Crane et al. in SP’15, J. Hiser et al. in SP’12), ISA (G. Portokalidis et al. in ACSAC’10, G. S. Kc et al. in CCS’03), Data Space Randomization (S. Bhatkar et al. in DIMVA’08) ...

**Bound Checkers** Address Sanitizer (K. Serebryany et al. in ATC’12), Cling (P. Akritidis et al. in SP’08), StackGuard (C. Cowan et al. in SEC’98), ...
Conclusion

A2C provides a general protection against a wide spectrum of payload injection attacks

- Malicious Input: program breaks, and *breaks early*
- Benign Input: program executes correctly

Key Idea: encodes inputs, decodes depending on the input processing semantics

A2C prevents payload injection with low overhead
Q&A

Thank you

Yonghwi Kwon
PhD student, Purdue University

Contact: yongkwon@purdue.edu
Web: http://yongkwon.info
More Slides

• Backup Slides
Evaluation
Decoding frontier computation

More decoding frontiers
71% of decoding frontiers turned out they are indeed decoding frontiers.

Exploitable-Space is *Small*
Inputs are quickly parsed and do not usually propagate deeply into a program. Exploitable-space is not huge which is a key reason of our low overhead.
Case Study
Preventing ROP Attacks

```c
void process_font_table(...) {
    ...
    char name[255];
    ...
    while (w2) {
        tmp = word_string(w2);
        if (tmp && DEC(tmp[0]) != '\\')
            strcat(name, tmp);
        ...
    }
}
```

---

<table>
<thead>
<tr>
<th>ROP Gadget</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804d820</td>
<td>mov ebx,0x0</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td>0x804ec7d</td>
<td>mov eax,0x806275c</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

... ... ...

**XOR 0xAA**

---

<table>
<thead>
<tr>
<th>ROP Gadget</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa2ae728a</td>
<td>Invalid address</td>
</tr>
<tr>
<td>0xa2ae46d7</td>
<td>Invalid address</td>
</tr>
</tbody>
</table>

... ... ...

Decoding/Encoding Sets
Static Analysis

Encoding Set: When to encode?
Encode Incoming Untrusted Sources at Library Calls (e.g., recv, read)

Decoding Set: When to decode?
Decode when encoded values are used
- Decode permanently at decoding frontier

Finding Decoding/Encoding Sets
Flow-, Context-, Field-sensitive Static Analysis
recv(..., untrusted_buf, ...); ENC(untrusted_buf);

...  

if ( DEC(untrusted_buf[0] ) == 'C' ) {

   ...

}  

...  

int ret = memcmp(DEC(untrusted_buf), ...);
Decoding is *not* simple

```c
recv(..., untrusted_buf, ...); ENC(untrusted_buf);
...
if (DEC(untrusted_buf[0]) == 'C') {
    memcpy(untrusted_buf, "CONSTANT", ...);
}
...
int ret = memcmp(DEC(untrusted_buf), ...,);
```

`untrusted_buf` can be from ‘recv’ and ‘constant’
Decoding is *not* simple

```c
recv(..., untrusted_buf...); ENC(untrusted_buf);
...
if (DEC(untrusted_buf[0]) == 'C') {
    memcpy(untrusted_buf, "CONSTANT", ...);
}
...;
int ret = memcmp(DEC(untrusted_buf), ...,);
```

Decoding *untrusted_buf* will break when it holds “CONSTANT”

Not Decoding *untrusted_buf* will break when its value is from `recv`
Decoding is not simple

recv(..., untrusted_buf, ...); ENC(untrusted_buf);
...
if (DEC(untrusted_buf[0]) == 'C') {
    memcpy(untrusted_buf, "CONSTANT", ...);
}
...
int ret = memcmp(DEC(untrusted_buf), ...);

We also encode "CONSTANT"

Now, decoding untrusted_buf will not break in any context.
Decoding is not simple

recv(..., untrusted_buf, ...); ENC( untrusted_buf );
...
if ( DEC( untrusted_buf[0] ) == 'C' ) {
    memcpy( untrusted_buf, ENC("CONSTANT"), ... );
}
...

int ret = memcmp( DEC( untrusted_buf ), ... );

untrusted_buf is always encoded in any context
### Evaluation

Different Types of Decoding Frontiers

1. Comparative:
   \[ x == y \]

2. Terminal:
   \[ \text{send}( x ) \]

3. Type widening:
   \[ \text{int } y = (\text{char})x; \]

4. Primitive Type Conversion:
   \[ \text{float } v = \text{atof}(x); \]

5. Indexing:
   \[ y = \text{array}[x]; \]
Evaluation
Decoding Frontier Computation

14 = Avg. Constraints
We mostly find that # of constraints for decoding frontier computation is not very large (10-20). This makes the fast computation possible.