Verifying Data-Oriented Gadgets in Binary Programs to Build Data-Only Exploits

Zachary Sisco

Wright State University

June 14, 2018

Introduction

Motivation

- *Data-only attacks*: manipulate program's data plane
- *Data-oriented programming*: expressive data-only attacks; chain together instructions to simulate computation

Introduction

Problem Statement

- This thesis explores the feasibility of constructing data-oriented programming exploits in binary programs without source code.
- Specifically: Classifying data-oriented gadgets and their properties.
- Why?
  - No current binary-based classification
  - For defense and security analysis (source not always available)

Introduction

Contributions

- A methodology for formally classifying data-oriented gadgets in binary programs without source code.
- A prototype implementation that shows prevalence of gadgets in binaries and demonstrates how data-only exploits can be crafted without source code.

Introduction

Outline

- Background: Data-only attacks, Data-oriented Programming
- Methodology and Implementation
- Evaluation and Results
- Conclusions

Background

Memory corruption vulnerabilities

- Buffer overflows, integer overflows, format string errors
- Attacks: Hijack control flow (overwrite return addresses, function pointers); execute malicious code
- Defenses: ASLR, DEP, Control-flow Integrity

Background

Data-only attacks [Chen et al., 2005]

- Manipulate data pointers (preserve control flow)
- Corrupt logic and decision-making routines

Data-only Attack Example

```c
struct passwd { uid_t pw_uid; ... } *pw;
...
int uid = getuid();
pw->pw_uid = uid;
printf(...); // format string vulnerability
...
seteuid(0); // set root id
setsockopt(...);
...
seteuid(pw->pw_uid); // set unprivileged user id
...
```

Listing 1: Vulnerable code snippet in `wu-ftpd`.

Background

Data-oriented Programming (DOP) [Hu et al., 2016]

- Expressive (Turing-complete) data-only attacks
- Data-oriented “gadgets”: simulate micro-operations
- Dispatchers: chains gadgets to perform arbitrary computation
- Still respects control flow
- Examples: SSL private key leak; bypass DEP and CFI

Research Objectives

Contributions

- A methodology for formally classifying data-oriented gadgets in binary programs without source code.
- A prototype implementation that shows prevalence of gadgets in binaries and demonstrates how data-only exploits can be crafted without source code.

Questions

- How does gadget classification differ between source and binary-based analysis?
- How does the compiler affect the type and frequency of gadgets?

Methodology

Overview

1. Identify potential data-oriented gadgets in a binary
2. Formally verify their semantics
3. Determine reachability of gadgets to vulnerable function

Methodology

Data-oriented gadgets

1. Ends with a Store instruction
2. At least one Load instruction
3. Gadget "body": semantics between Load and Store

Methodology

Data-oriented gadgets

```
*p += *q; /* p, q are (int*) type */
```

```
mov eax, DWORD PTR [ebp-0xC]        ;load p to eax
mov edx, DWORD PTR [eax]            ;load *p to edx
mov eax, DWORD PTR [ebp-0x10]       ;load q to eax
mov eax, DWORD PTR [eax]            ;load *q to eax
add edx, eax                        ;add *q to *p
mov eax, DWORD PTR [ebp-0xC]        ;load p to eax
mov DWORD PTR [eax], edx            ;store edx in *p
```

Figure 1: Example showing a snippet of C code and the corresponding X86 assembly instructions.

Methodology

Identifying Data-oriented Gadgets

1. Lift to Intermediate Representation (IR)
2. Backward static program slicing to find gadget instructions

Methodology

Identifying Data-oriented Gadgets

1. Lift to Intermediate Representation (IR)
   - VEX-IR through angr framework
   - SSA
   - Abstracts architectural differences
2. Backward static program slicing to find gadget instructions

Methodology

Backward Static Program Slicing

Given a program P, a backward program slice at program point p with set of variables V contains only those preceding statements in P that affect the variables in V at p [Weiser, 1981].

Methodology

```
t55 = LDle:I32(0x080499a8)
PUT(offset=68) = 0x08048597
t56 = LDle:I32(0x080499a8)
PUT(offset=68) = 0x0804859d
t57 = LDle:I32(t56)
PUT(offset=12) = t57
PUT(offset=68) = 0x0804859f
t58 = LDle:I32(0x080499ac)
PUT(offset=68) = 0x080485a5
t59 = LDle:I32(t58)
t24 = Add32(t59,t57)
PUT(offset=40) = 0x00000003
PUT(offset=44) = t59
PUT(offset=48) = t57
PUT(offset=52) = 0x00000000
PUT(offset=16) = t24
PUT(offset=68) = 0x080485a9
STle(t55) = t24
```

Methodology

```
t55 = LDle:I32(0x080499a8)
PUT(offset=68) = 0x08048597
t56 = LDle:I32(0x080499a8)
PUT(offset=68) = 0x0804859d
t57 = LDle:I32(t56)
PUT(offset=12) = t57
PUT(offset=68) = 0x0804859f
t58 = LDle:I32(0x080499ac)
PUT(offset=68) = 0x080485a5
t59 = LDle:I32(t58)
t24 = Add32(t59,t57)
PUT(offset=40) = 0x00000003
PUT(offset=44) = t59
PUT(offset=48) = t57
PUT(offset=52) = 0x00000000
PUT(offset=16) = t24
PUT(offset=68) = 0x080485a9
STle(t55) = t24
```

Methodology

Backward Static Program Slicing

```
t55 = LDle:I32(0x080499a8)
t56 = LDle:I32(0x080499a8)
t57 = LDle:I32(t56)
t58 = LDle:I32(0x080499ac)
t59 = LDle:I32(t58)
t24 = Add32(t59,t57)
STle(t55) = t24
```

- `t55` is the *address* parameter for Store
- `t24` is the *data* parameter for Store

Methodology

Overview

1. Identify potential data-oriented gadgets in a binary
2. Formally verify their semantics
3. Determine reachability of gadgets to vulnerable function

Methodology

Program Verification for Gadget Semantics

- For a program S,
  If a gadget is of type described by Q, a first-order predicate, then after executing the statements in S the program is in a state satisfying Q.
- The **Weakest Precondition**, $wp(S, Q)$, is a predicate that characterizes all initial states of S such that it terminates in a final state satisfying Q [Dijkstra, 1976].

Deriving Weakest Preconditions from Gadgets

$$
\begin{array}{lcll}
\text{GCL } \textit{Statement} \quad s & ::= & x := e \\
& | & \textbf{assume } e \\
& | & s \; ; \; s \\
& | & s \;\square\; s
\end{array}
$$

$$\frac{}{wp(x := e, Q) : Q[e/x]}\text{WP-ASSIGN}$$

$$\frac{}{wp(\textbf{assume } e, Q) : e \Rightarrow Q}\text{WP-ASSUME}$$

$$\frac{wp(s_2, Q) : Q_1 \quad wp(s_1, Q_1) : Q_2}{wp(s_1 \; ; \; s_2, Q) : Q_2}\text{WP-SEQUENCE}$$

$$\frac{wp(s_1, Q) : Q_1 \quad wp(s_2, Q) : Q_2}{wp(s_1 \square s_2, Q) : Q_1 \wedge Q_2}\text{WP-CHOICE}$$

Figure 2: Dijkstra's Guarded Command Language (GCL) and Predicate Transformers.

Methodology

Characterizing Semantics

Name	Parameters	Postcondition
Move	**Out**, **In**	$\textbf{Out} = \textbf{In}$
Load	**Out**, **In**	$\textbf{Out} = \mathcal{M}[\textbf{In}]$
Store	**Out**, **In**	$\mathcal{M}[\textbf{Out}] = \textbf{In}$
Arithmetic	**Out**, x, y	$\textbf{Out} = x \diamond_a y$
Logical	**Out**, x, y	$\textbf{Out} = x \diamond_\ell y$
Conditional	**Out**, x, y	$((x \diamond_c y) \Rightarrow \textbf{Out} = 1) \wedge$ $(\neg(x \diamond_c y) \Rightarrow \textbf{Out} = 0)$

Table 1: Postconditions for verifying data-oriented gadget semantics. $\diamond_a$ is an arithmetic binary operator; $\diamond_\ell$ is a logical binary operator; and $\diamond_c$ is a comparison operator.

Methodology

Finding Gadget Parameters

```
t33 = GET:I32(offset=28) # 28 is EBP
t35 = Add32(t33, 0xffffffe0)
t37 = LDle:I32(t35)
t38 = LDle:I32(t37)
t34 = LDle:I32(0x805c7e8)
STle(t34) = t38
```

Listing 2: VEX-IR example program slice demonstrating two examples of variable scope inference in VEX-IR. `t34` is a global variable, and `t38` is a local variable.

Methodology

Overview

1. Identify potential data-oriented gadgets in a binary
2. Formally verify their semantics
3. Determine reachability of gadgets to vulnerable function
   - Dynamic function trace in presence of vulnerability (Intel PIN)

Methodology

Implementation

- python
- angr — binary analysis framework
- Z3 — SMT solver

Limitation

- Simple vs complex gadgets

Evaluation

Setup

- Compare results with source-based analysis by [Hu et al., 2016]
- Intel x86 32-bit, Debian 8.10 on Linux kernel version 3.16.
- Programs compiled with GCC 4.9.2 and Clang 3.5.0
- Selected programs: curl, imlib2, libtiff, nginx, optipng, sudo, unzip

Classification Results

Application	Version	Binary/Source	Compiler	Dispatchers	Assign			Deref			Arith			Logic			Cond		
					G	H	L	G	H	L	G	H	L	G	H	L	G	H	L
curl	7.41.0	B	GCC	71	99	143	27	62	25	559	36	303	16	5	0	0	1	0	0
		B	Clang	76	349	311	1	87	27	53	16	318	1	6	2	0	3	0	0
		S	Clang	11	0	2	15	0	2	26	2	0	5	1	2	17	0	0	4
imlib2	1.4.7	B	GCC	734	440	275	0	220	101	633	1208	256	137	65	19	15	4	4	0
		B	Clang	835	934	721	3	180	86	65	262	223	22	33	29	9	0	0	0
		S	Clang	152	1	5	55	25	183	94	12	96	390	3	219	411	3	1	19
libtiff	2.5.6	B	GCC	358	264	309	7	226	99	578	328	161	52	6	16	19	0	2	0
		B	Clang	374	451	290	35	87	175	5	148	179	5	15	11	1	1	0	0
		S	Clang	116	2	9	83	0	62	333	1	79	243	0	35	183	0	5	20
nginx	1.4.0	B	GCC	499	1316	526	1	333	190	87	276	440	9	97	2	4	3	0	0
		B	Clang	496	1104	497	11	378	328	114	225	426	14	83	11	3	4	0	0
		S	Clang	206	12	55	74	32	40	958	12	26	156	28	7	290	3	7	22
optipng	0.7.6	B	GCC	219	167	165	4	45	76	208	183	260	45	24	5	1	3	2	0
		B	Clang	249	244	114	13	19	42	38	144	235	8	23	1	1	1	0	0
		S	Clang	63	35	35	73	10	1	245	11	35	283	17	25	146	4	4	69
sudo	1.8.3p1	B	GCC	91	129	123	127	10	45	92	21	317	101	11	0	9	3	0	0
		B	Clang	65	103	44	1	13	15	12	14	239	0	12	0	0	3	0	0
		S	Clang	16	11	0	9	11	5	8	3	0	9	5	2	0	3	0	0
unzip	6.0	B	GCC	209	133	204	32	68	43	136	194	218	13	16	12	2	1	0	0
		B	Clang	182	215	32	0	21	11	12	161	139	2	8	0	3	1	0	0
		S	Clang	28	45	14	4	146	6	1	147	4	7	72	2	0	34	2	0

Table 2: For gadget scopes, ‘G’ is Global, ‘H’ is Hybrid (mixed between global and local), and ‘L’ is Local.

Reachability Results

Application	CVE	Compiler	Dispatchers	Assign			Deref			Arith			Logic			Cond		
				G	H	L	G	H	L	G	H	L	G	H	L	G	H	L
curl	2015-3144[1]	GCC	5	0	4	0	0	1	1	0	52	0	0	0	0	0	0	0
		Clang	8	0	4	0	0	0	2	0	56	0	0	0	0	0	0	0
imlib2	2016-3994[2]	GCC	12	7	6	0	7	3	17	3	36	0	0	0	4	0	0	0
		Clang	14	18	7	0	0	6	6	1	28	0	0	0	0	0	0	0
libtiff	2017-9935[3]	GCC	16	13	26	0	13	0	48	5	36	11	1	0	1	0	0	0
		Clang	14	16	9	0	5	3	2	7	9	0	2	0	0	0	0	0
nginx	2013-2028[4]	GCC	69	370	136	1	54	49	19	71	49	0	50	0	0	2	0	0
		Clang	77	297	110	0	80	51	18	44	44	0	28	3	1	1	0	0
optipng	2016-3982[5]	GCC	3	0	3	1	0	0	0	0	28	0	0	0	0	0	0	0
		Clang	4	0	0	0	0	0	0	0	24	0	0	0	0	0	0	0
sudo	2012-0809[6]	GCC	5	23	4	1	2	0	2	0	0	0	0	0	0	1	0	0
		Clang	9	20	0	0	1	0	2	1	0	0	12	0	0	0	0	0
unzip	2015-7696[7]	GCC	6	0	0	0	0	0	0	0	60	0	0	0	0	0	0	0
		Clang	6	0	0	0	0	0	0	0	48	0	0	0	0	0	0	0

Table 3: Data-oriented gadget reachability results with respect to a vulnerable function trace through a reported vulnerability from the CVE database. For gadget scopes, ‘G’ is Global, ‘H’ is Hybrid, and ‘L’ is Local.

nginx Exploit

```
for ( ;; ) {
    rc = ngx_http_parse_chunked(r, b, rb->chunked);
...
    if (rc == NGX_AGAIN) {
        /* Two dereferenced assignment gadgets */
        r->headers_in.content_length_n =
            rb->chunked->length;
        break;
    }
...
```

Listing 3: Vulnerable code snippet in the function `ngx_http_discard_request_body_filter` in “nginx.”

nginx Exploit

1. Send chunked HTTP request ($\geq$1024 bytes) to set `rc` to `NGX_AGAIN`.
   (This also sets `rb->chunked->length` to a large number.)
2. The data-oriented gadgets execute in line 6.
   (`r->headers_in.content_length_n` becomes negative.)
3. `ngx_http_parse_chunked` executes a second time.
   Send $\geq$4096 bytes, overflowing a vulnerable buffer on the stack.

Conclusions

The Problem

- Classify data-oriented gadgets in binary
- Feasibility of constructing DOP exploits without source

Findings, Limitations, Implications

- Classification differs between source and binary-based analysis
- How compilers emit code also affects gadget discovery
- Prototype supports classification for binaries under any compiler
- For security, expands the range of software that can be analyzed

Conclusions

Future Work

- Complex gadget classification
- Formally verify gadget properties
- Automating DOP exploit construction

Thank you

- Questions

References I

Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. (2005).
Non-control-data attacks are realistic threats.
In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 1–15, Berkeley, CA, USA. USENIX Association.

Dijkstra, E. W. (1976).
A Discipline of Programming.
Prentice Hall PTR, Englewood Cliffs, NJ, USA, 1st edition.

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. (2016).
Data-oriented programming: On the expressiveness of non-control data attacks.
In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986.

References II

Weiser, M. (1981).
Program slicing.
In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press.

Appendix

---

function: SetRelevantVariables(*B*, *relevantVariables*)
input: *B*, basic block; *relevantVariables*, maps a statement to a set of variables
output: *relevantVariables*

```
foreach statement i and successor statement j in B:
        if i.LHS in relevantVariable[j] then:
            // add all variables used by i to the relevant variables of i
            relevantVariables[i].add(i.variables)
        else:
            // add that variable to the relevant variables of j
            relevantVariables[j].add(i.LHS)
```

Appendix

---

function: BackwardProgramSlice(B, *relevantVariables*)
input: B, basic block; *relevantVariables*, maps a statement to a set of variables
output: program slice, set of statements

relevantVariables $\leftarrow$ SetRelevantVariables(B, *relevantVariables*)
foreach statement i **and** successor statement j **in** B:
    if i.LHS **in** *relevantVariables*[j] **then**:
        add i to the program slice

Appendix

```
function: BDFA(v, slice, istack)
input: v, target variable ; slice, program slice
output: istack, instruction stack tracing v

if slice is empty then:
    return
i ← slice.pop()
if i is Assignment Instruction then:
    if i.LHS = v then:
        istack.push(i)
        rhs ← GetVariables(i.RHS)
        foreach variable t in rhs:
            BDFA(t, slice, istack)
else :
    BDFA(v, slice, istack)
```

Appendix

function: GetGadget(*store*, B)
input: *store*, a Store Instruction; B, basic block
output: a pair of instruction stacks

$relevantVariables \leftarrow \emptyset$
$addrInstr \leftarrow \emptyset$
$dataInstr \leftarrow \emptyset$

$relevantVariables[store]$.add($store$.addr)
$relevantVariables[store]$.add($store$.data)
$progSlice \leftarrow$ BackwardProgramSlice(B, $relevantVariables$)
$addrInstr \leftarrow$ BDFA($store$.addr, $progSlice$, $addrInstr$)
$dataInstr \leftarrow$ BDFA($store$.data, $progSlice$, $dataInstr$)
// Potential gadgets must have at least one Load instruction
if Load Instruction **in** *addrInstr* **or** *dataInstr* **then**:
    return $\langle addrInstr, dataInstr \rangle$

Appendix

---

function: GetPotentialGadgets(*prog*)
input: *prog*, program **in** VEX−IR
output: *potentialGadgets*, a list of pairs of instruction stacks

```
foreach func in prog:
    foreach loop in func:
        foreach basic block B in loop:
            foreach stmt in B:
                if stmt is Store Instruction then:
                    g ← getGadget(stmt, B)
                    potentialGadgets.add(g)
                if stmt is Call Instruction then:
                    target ← followCallGraph(stmt)
                    foreach stmt in target:
                        if stmt is Store Instruction then:
                            g ← getGadget(stmt, target)
                            potentialGadgets.add(g)
```

Appendix

```
t5 = LDle:I32(0x0805c7e8)
t1 = Add32(t5,0x00000002)
STle(0x0805c7e8) = t1
```

Listing 4: VEX-IR example program slice demonstrating an arithmetic ADD gadget.

$$\begin{aligned} \mathcal{T}[5] &:= \mathcal{M}[\mathtt{0x805c7e8}]; \\ \mathcal{T}[1] &:= \mathcal{T}[5] + 2; \\ \mathcal{M}[\mathtt{0x805c7e8}] &:= \mathcal{T}[1] \end{aligned} \tag{1}$$

Appendix

$$
\begin{aligned}
wp(s_1; s_2, wp(s_3, Q)) &= wp(s_1; s_2, \mathcal{T}[1] = \mathcal{T}[5] + 2) \\
&= wp(s_1, wp(s_2, \mathcal{T}[1] = \mathcal{T}[5] + 2)) \\
&= wp(s_1, \mathcal{T}[5] + 2 = \mathcal{T}[5] + 2) \\
&= (\mathcal{M}[\mathtt{0x805c7e8}] + 2 = \mathcal{M}[\mathtt{0x805c7e8}] + 2)
\end{aligned}
$$

Figure 3: Application of weakest precondition derivation rules. The initial value of Q is $(\mathcal{M}[\mathtt{0x805c7e8}] = \mathcal{T}[5] + 2)$.

Appendix

Application	Compiler	Gadget Length				Gadget Parameters				Gadgets per Function				Gadgets per Dispatcher			
		Mean	Median	Min	Max	Mean	Median	Min	Max	Mean	Median	Min	Max	Mean	Median	Min	Max
curl	GCC	7	7	3	18	2	2	2	4	9	1	4	553	27	4	1	672
	Clang	5	5	2	17	2	2	2	4	9	4	1	495	27	5	1	592
imlib2	GCC	7	7	4	29	2	2	2	5	11	4	1	223	12	7	1	527
	Clang	6	6	4	17	2	2	2	4	7	2	1	104	9	2	1	673
libtiff	GCC	8	7	4	23	2	2	2	5	7	3	1	257	9	3	1	539
	Clang	7	6	4	18	2	2	2	6	8	4	1	185	7	2	1	124
nginx	GCC	6	6	3	26	2	2	2	6	7	4	1	93	10	4	1	471
	Clang	6	6	3	26	2	2	2	6	7	4	1	107	9	3	1	448
optipng	GCC	7	7	4	33	3	2	2	9	7	4	1	96	11	3	1	254
	Clang	6	6	3	31	2	2	2	10	8	4	1	69	6	2	1	122
sudo	GCC	6	5	2	18	2	2	2	4	7	4	1	75	14	4	1	126
	Clang	5	5	2	14	3	3	2	4	5	4	1	48	11	5	1	58
unzip	GCC	7	6	4	28	2	2	2	7	10	4	1	81	12	2	1	296
	Clang	5	4	2	27	2	2	2	7	8	4	1	109	10	2	1	112

Table 4: Data-oriented gadget statistics.

Appendix

Application	Compiler	Registers	Stack	Constant
curl	GCC	2519	485	0
	Clang	1525	1207	16
imlib2	GCC	9005	8319	0
	Clang	8216	8742	0
libtiff	GCC	4203	2797	0
	Clang	2993	2940	0
nginx	GCC	5176	3559	93
	Clang	4904	3629	93
optipng	GCC	3558	1560	0
	Clang	2236	1076	34
sudo	GCC	1208	1266	102
	Clang	948	271	62
unzip	GCC	2348	1373	0
	Clang	828	497	367

Table 5: Comparison of parameter-loading strategies for gadgets in each program compiled under GCC and Clang.

Appendix

Application	Compiler	Verified Gadget Count	Potential Complex Gadget Count	Potential Gadget Total	Verified Gadget Proportion	Complex Gadget Proportion
curl	GCC	1276	14	1298	98%	1%
	Clang	1174	13	1197	98%	1%
imlib2	GCC	3377	1516	5048	67%	30%
	Clang	2567	1976	4717	54%	42%
libtiff	GCC	2067	410	2572	80%	16%
	Clang	1403	479	1985	71%	24%
nginx	GCC	3284	285	3697	89%	8%
	Clang	3198	259	3615	88%	7%
optipng	GCC	1188	288	1582	75%	18%
	Clang	883	193	1125	78%	17%
sudo	GCC	988	49	1056	94%	5%
	Clang	456	22	486	94%	5%
unzip	GCC	1072	228	1350	79%	17%
	Clang	605	100	722	84%	14%

Table 6: Comparison of verified data-oriented gadget totals and potential complex gadgets that are omitted. “Potential Gadget Total” includes all instruction sequences considered for classification.

Appendix

```
# Binary; compiled with GCC
t9 = GET:I32(offset=28)
t8 = Add32(t9,0xffffffec)
t10 = LDle:I32(t8)
t11 = Add32(t10,0x00000004)
t13 = LDle:I32(t11)
t15 = Add32(t9,0xfffffff4)
t3 = LDle:I32(t15)
t2 = Add32(t13,t3)
t17 = Add32(t9,0xffffffec)
t19 = LDle:I32(t17)
t20 = Add32(t19,0x00000004)
STle(t20) = t2
```

```
# Source; compiled with Clang
%iob.028 = load %struct.io_buffer** @iobufs
%iob.030 = phi %struct.io_buffer* [ %iob.0, %.loopexit ],
           [ %iob.028, %.lr.ph.preheader ]
%13 = getelementptr inbounds %struct.io_buffer* %iob.030,
      i32 0, i32 1
%48 = load i32* %13
%49 = add nsw i32 %48, %.lcssa
store i32 %49, i32* %13
```

Figure 4.2: Dereferenced arithmetic gadgets in the `perform_io` function in “sudo”.

Appendix

```
# Binary; compiled with GCC
t15 = GET:I32(offset=24)
t14 = Add32(t15,0x00000084)
t16 = LDle:I32(t14)
t5 = LDle:I32(t16)
t3 = Add32(t5,0x00000001)
STle(t16) = t3
```

```
# Binary; compiled with Clang
t4 = LDle:I32(0x0813c298)
t5 = Add32(t4,0x00000001)
STle(0x0813c298) = t5
```

```
# Source; compiled with Clang
%86 = load i32* getelementptr inbounds
      (%struct.Globals* @G, i32 0, i32 0, i32 7)
%91 = add nsw i32 %86, 1
store i32 %91, i32* getelementptr inbounds
      (%struct.Globals* @G, i32 0, i32 0, i32 7)
```

Figure 4.4: Arithmetic gadgets in “unzip”.

Appendix

```
# Gadget 1
t9 = GET:I32(offset=32)
t8 = Add32(t9,0x0000001c)
t10 = LDle:I32(t8)
t11 = Add32(t10,0x00000010)
t13 = LDle:I32(t11)
t18 = GET:I32(offset=36)
t17 = Add32(t18,0x000000e0)
STle(t17) = t13
```

```
# Gadget 2
t9 = GET:I32(offset=32)
t8 = Add32(t9,0x0000001c)
t10 = LDle:I32(t8)
t14 = Add32(t10,0x0000000c)
t16 = LDle:I32(t14)
t18 = GET:I32(offset=36)
t20 = Add32(t18,0x000000dc)
STle(t20) = t16
```

Figure 4.5: Two dereferenced assignment gadgets in "nginx."