Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage

Stephen Checkoway,[*] Ariel J. Feldman,[†] Brian Kantor,[*]
J. Alex Halderman,[‡] Edward W. Felten,[†] Hovav Shacham[*]

[*]UCSD, [†]Princeton, [‡]U Michigan

Voting System Studies

Study	Vendors	Year
Appel et al.	Sequoia	2008
EVEREST	ES&S, Hart, Premier	2007
California TTBR	Hart, Premier, Sequoia	2007
Feldman et al.	Diebold	2006
Hursti	Diebold	2006
Kohno et al.	Diebold	2003

Response

The proposed 'red team' concept also contemplates giving attackers access to source code, which is unrealistic and dangerous if not strictly controlled by test protocols. It is the considered opinion of election officials and information technology professionals that ANY system can be attacked if source code is made available. We urge the Secretary of State not to engage in any practice that will jeopardize the integrity of our voting systems.

– California Association of Clerks and Election Officials, 2007

Response

No computer system could pass the assault made by your team of computer scientists. In fact, I think my 9 and 12-year-old kids could find ways to break into the voting equipment if they had unfettered access.

– Santa Cruz County Clerk Gail Pellerin, 2007

Is it practical to hack a voting machine without "unreasonable" access?

Hint: Yes

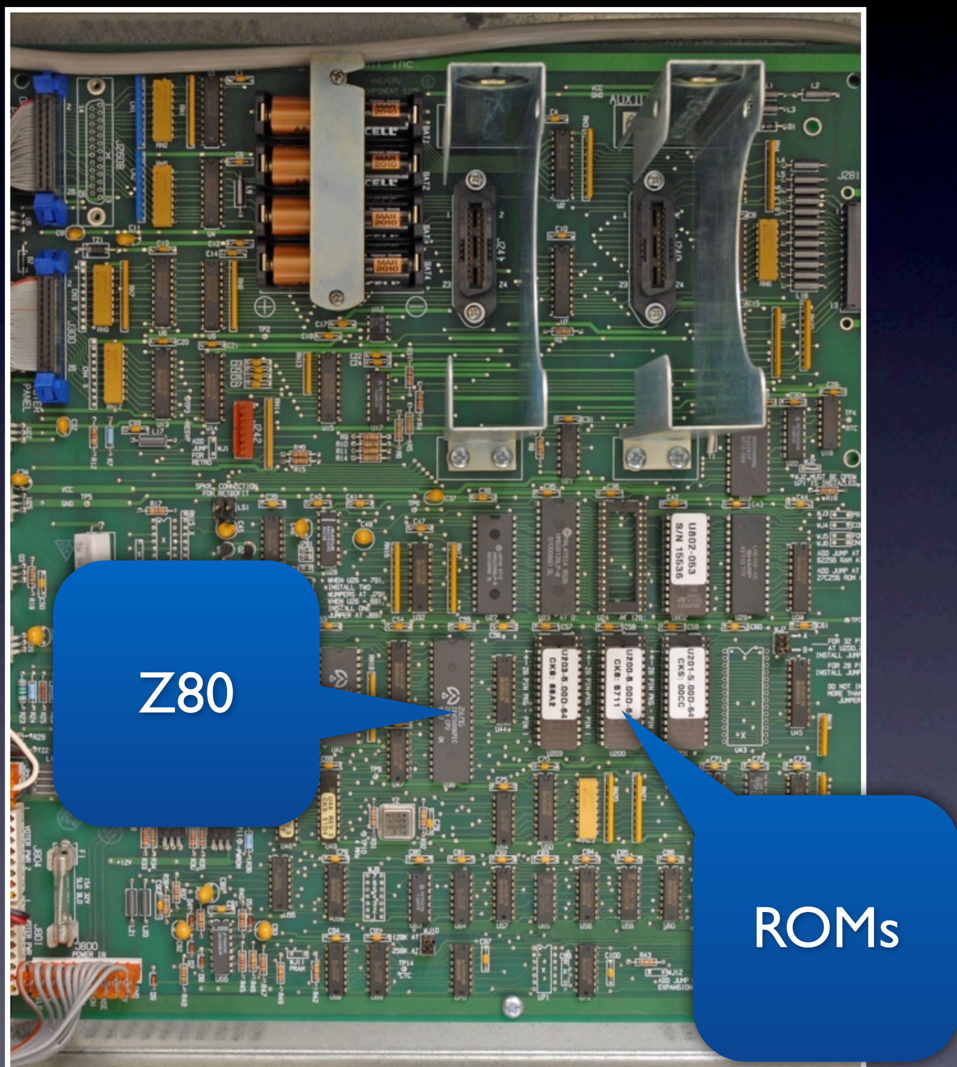
AVC Advantage

- Best-case to study
  - Only does one thing: count votes
  - Defenses against code injection

Challenges

1. Understand how the machine works without source code or documentation by reverse-engineering
2. Find an exploitable bug
3. Defeat code-injection defense using recently developed techniques from system security

Reverse-Engineering




```
; =============== S U B R O U T I N E ===============

; memcopy( from, to, size )
; returns 1 in bc on success and 0 if size = 0

memcopy:                                        ; CODE XREF:
                ld      hl, 2
                add     hl, sp
                ld      e, (hl)
                inc     hl
                ld      d, (hl)
                push    de                      ; push from
                inc     hl
                ld      e, (hl)
                inc     hl
                ld      d, (hl)                 ; de <- to
                inc     hl
                ld      c, (hl)
                inc     hl
                ld      b, (hl)                 ; bc <- size
                pop     hl                      ; hl <- from
                ld      a, b
                or      c
                jr      z, zero_copy            ; if bc = 0 t
                ldir                            ; copy bc by
                ld      bc, 1

zero_copy:                                      ; CODE XREF:
                ret
; End of function memcopy
```

Artifacts Produced

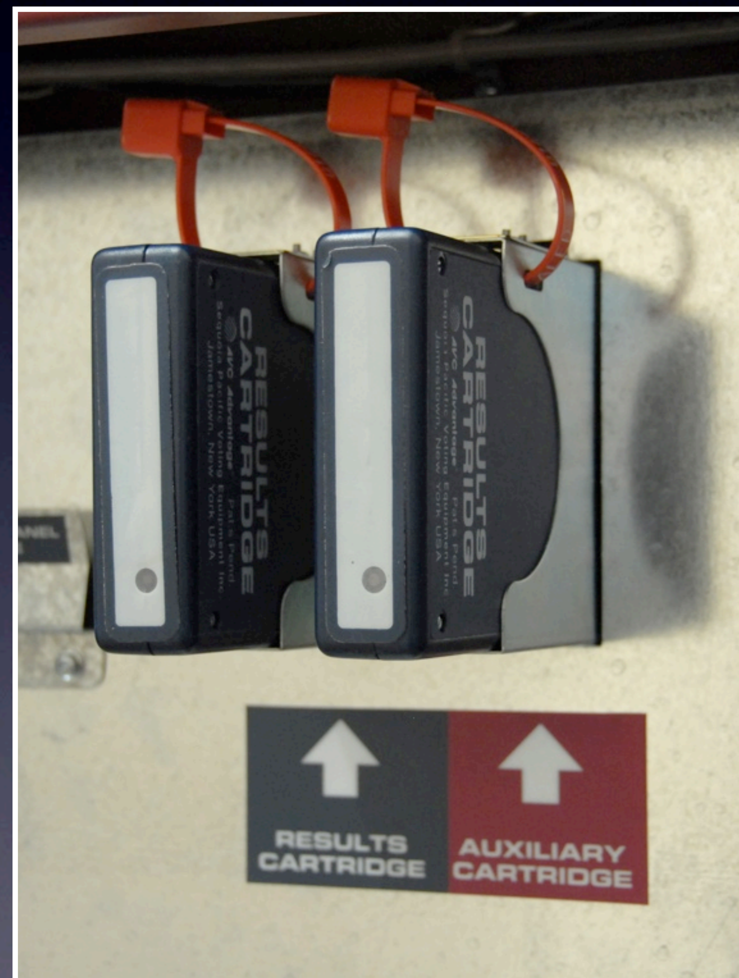
- Hardware Functional Specifications
- Hardware Simulator
  - Initial version by Joshua Herbach
  - Exploit developed on the simulator — tested on machine, worked first try

Exploit

- Classic stack-smashing buffer overflow
  - Roughly a dozen bytes overwritten
  - Exploit code needs to be in memory
- For now, assume we can inject code

Vote-Stealing Attack

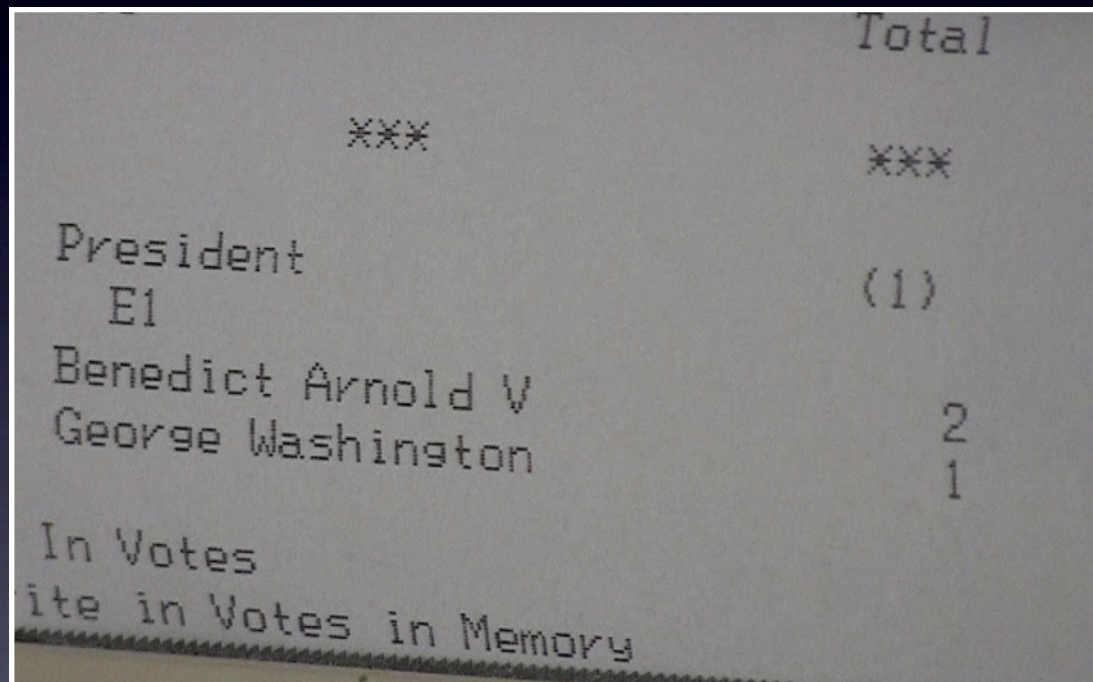
- Gain physical access
- Malicious auxiliary cartridge
- Trigger exploitable bug
- Follow instructions



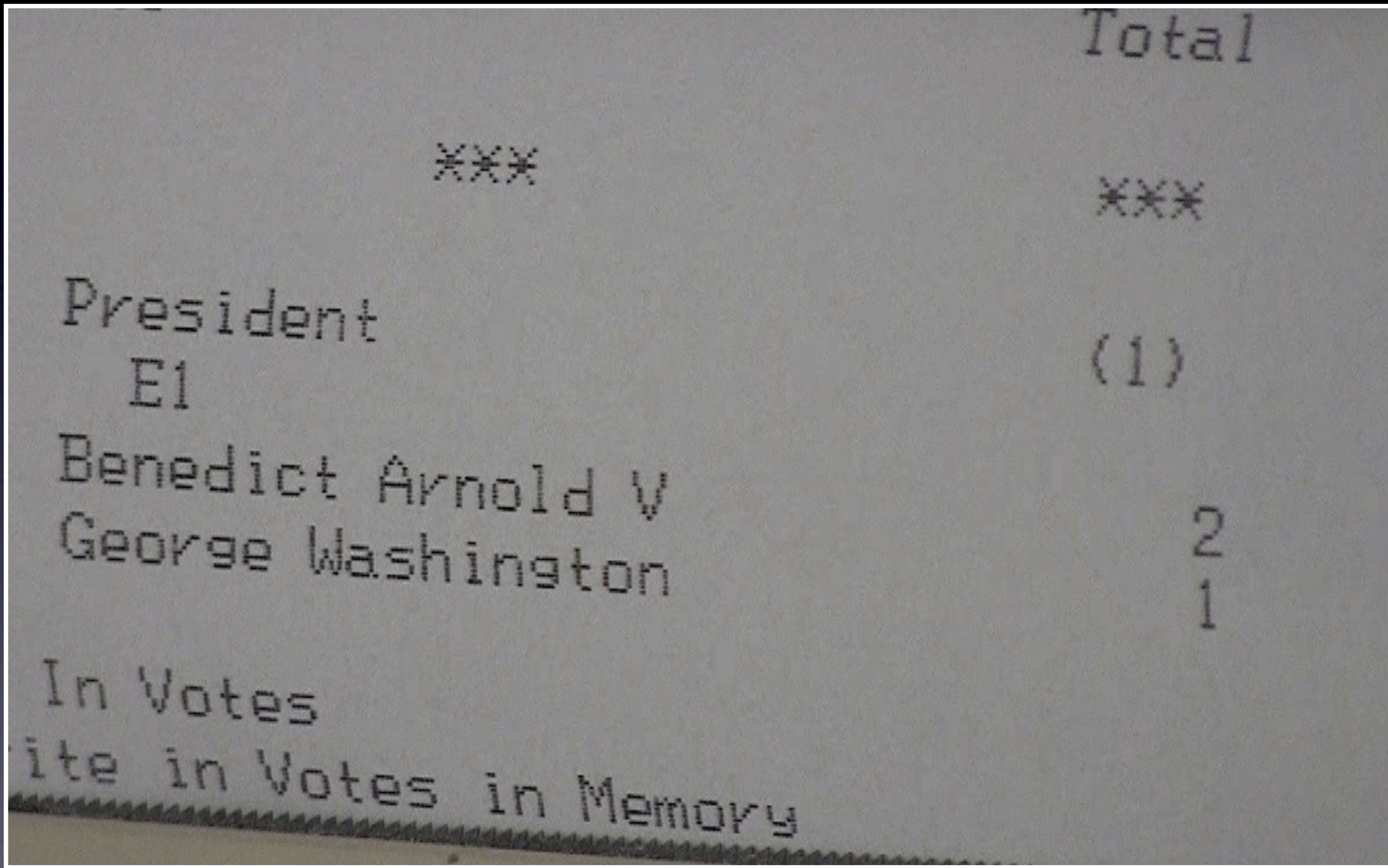

AVC
Advantage
TEST
Remove sploit cart
Turn power off
OPERATOR PANEL
1

Vote-Stealing Program

- Survives turning power switch to off
- Runs election as normal
- Silently shifts votes



Vote-Stealing Program



Code Injection?

- Earlier, we assumed we could inject code
- Hardware interlock prevents fetching instructions from RAM
- Program code in read-only memory

Harvard Architecture

Program in read-only memory

+

Nonexecutable, writable data memory

No code injection

Return-Oriented Programming

Return-Oriented Programming

- Arbitrary behavior without code injection
- Combine snippets of existing code
- Requires control of the call stack
- Processor/program specific

Return-Oriented Programming

Instructions

```
movl    $0x006f6d2e,(%eax,%ebx)
movl    0xd4(%ebp),%eax
movl    %eax,(%esp)
calll   0x0008ba11
addl    $0x1f,%eax
andl    $0xf0,%eax
subl    %eax,%esp
leal    0x20(%esp),%edx
movl    %edx,0xb4(%ebp)
jmp     0x0006d8b4
incl    0xd4(%ebp)
movl    0xd4(%ebp),%eax
movzbl  (%eax),%ecx
cmpb    $0x3a,%cl
je      0x0006d8b1
testb   %cl,%cl
movl    0xb4(%ebp),%ebx
jne     0x0006d8db
movb    $0x43,(%ebx)
movb    $0x00,0x01(%ebx)
jmp     0x0006d90d
movb    %cl,(%ebx)
incl    %ebx
incl    0xd4(%ebp)
movl    0xd4(%ebp),%eax
movzbl  (%eax),%ecx
testb   %cl,%cl
setne   %dl
cmpb    $0x3a,%cl
setne   %al
testb   %al,%dl
jne     0x0006d8cf
movb    $0x00,(%ebx)
cmpl    $0x01,0x0008a780
jne     0x0006d90d
movl    0xb4(%ebp),%edx
movl    $0x0000002f,0x04(%esp)
movl    %edx,(%esp)
calll   0x0008b9e9
testl   %eax,%eax
jne     0x0006d8b4
movl    0xb4(%ebp),%esi
movl    $0x00000002,%ecx
movl    $0x0007e270,%edi
cld
repz/cmpsb      (%esi),(%edi)
movl    $0x00000000,%eax
je      0x0006d92e
movzbl  0xff(%esi),%eax
movzbl  0xff(%edi),%ecx
subl    %ecx,%eax
testl   %eax,%eax
jel     0x0006da53
movl    0xb4(%ebp),%esi
movl    $0x00070bbb,%edi
movl    $0x00000006,%ecx
repz/cmpsb      (%esi),(%edi)
movl    $0x00000000,%edx
je      0x0006d956
movzbl  0xff(%esi),%edx
movzbl  0xff(%edi),%ecx
subl    %ecx,%edx
testl   %edx,%edx
```

- Arbitrary behavior without code injection
- Combine snippets of existing code
- Requires control of the call stack
- Processor/program specific

Return-Oriented Programming

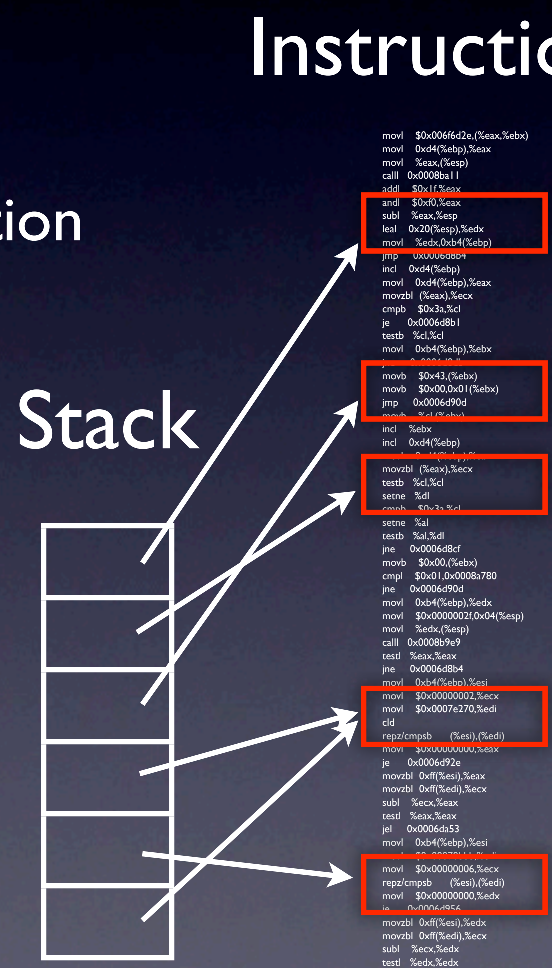
Instructions

- Arbitrary behavior without code injection
- Combine snippets of existing code
- Requires control of the call stack
- Processor/program specific

```
movl    $0x006f6d2e,(%eax,%ebx)
movl    0xd4(%ebp),%eax
movl    %eax,(%esp)
calll   0x0008ba11
addl    $0x1f,%eax
andl    $0xf0,%eax
subl    %eax,%esp
leal    0x20(%esp),%edx
movl    %edx,0xb4(%ebp)
jmp     0x0006d8b4
incl    0xd4(%ebp)
movl    0xd4(%ebp),%eax
movzbl  (%eax),%ecx
cmpb    $0x3a,%cl
je      0x0006d8b1
testb   %cl,%cl
movl    0xb4(%ebp),%ebx
movb    $0x43,(%ebx)
movb    $0x00,0x01(%ebx)
jmp     0x0006d90d
incl    %ebx
incl    0xd4(%ebp)
movzbl  (%eax),%ecx
testb   %cl,%cl
setne   %dl
setne   %al
testb   %al,%dl
jne     0x0006d8cf
movb    $0x00,(%ebx)
cmpl    $0x01,0x0008a780
jne     0x0006d90d
movl    0xb4(%ebp),%edx
movl    $0x0000002f,0x04(%esp)
movl    %edx,(%esp)
calll   0x0008b9e9
testl   %eax,%eax
jne     0x0006d8b4
movl    0xb4(%ebp),%esi
movl    $0x00000002,%ecx
movl    $0x0007e270,%edi
cld
repz/cmpsb      (%esi),(%edi)
je      0x0006d92e
movzbl  0xff(%esi),%eax
movzbl  0xff(%edi),%ecx
subl    %ecx,%eax
testl   %eax,%eax
jel     0x0006da53
movl    0xb4(%ebp),%esi
movl    $0x00000006,%ecx
repz/cmpsb      (%esi),(%edi)
movl    $0x00000000,%edx
movzbl  0xff(%esi),%edx
movzbl  0xff(%edi),%ecx
subl    %ecx,%edx
testl   %edx,%edx
```

Return-Oriented Programming

- Arbitrary behavior without code injection
- Combine snippets of existing code
- Requires control of the call stack
- Processor/program specific



The Usual Method

High-level specification

```
if arnold ≤ washington:
      amount = (washington - arnold)/2 + 1
      arnold = arnold + amount
      washington = washington - amount
```

The Usual Method

High-level specification

```
if arnold ≤ washington:
    amount = (washington - arnold)/2 + 1
    arnold = arnold + amount
    washington = washington - amount
```

Assembly

```
movl    ...., %edx
movl    ...., %ecx
compl   %ecx, %edx
jg      winning
movl    %ecx, %eax
subl    %edx, %eax
shrl    %eax
incl    %eax
addl    %eax, %edx
movl    %edx, ...
subl    %eax, %ecx
movl    %ecx, ...
winning:
```

The Usual Method

High-level specification

```
if arnold ≤ washington:
    amount = (washington - arnold)/2 + 1
    arnold = arnold + amount
    washington = washington - amount
```

Assembly

```
movl    ...., %edx
movl    ...., %ecx
compl   %ecx, %edx
jg      winning
movl    %ecx, %eax
subl    %edx, %eax
shrl    %eax
incl    %eax
addl    %eax, %edx
movl    %edx, ...
subl    %eax, %ecx
movl    %ecx, ...
winning:
```

Binary

```
00000000 55 89 e5 53 e8 00 00 00 00 5b 8b 93 2f 00 00 00
00000010 8b 8b 2b 00 00 00 39 ca 77 17 89 c8 29 d0 d1 e8
00000020 40 01 c2 89 93 2f 00 00 00 29 c1 89 8b 2b 00 00
00000030 00 5b c9 c3
```

The ROP Method

High-level specification

```
if arnold ≤ washington:
    amount = (washington - arnold)/2 + 1
    arnold = arnold + amount
    washington = washington - amount
```

The ROP Method

High-level specification

```
if arnold ≤ washington:
        amount = (washington - arnold)/2 + 1
        arnold = arnold + amount
        washington = washington - amount
```

Pseudo-assembly

```
ld      t1, 0(A)
ld      t2, 2(A)
slt     t3, t2, t1
btr     t3, winning
sub     amt, t2, t1
srl     amt, amt, 1
inc     amt
sub     t2, t2, amt
add     t1, t1, amt
st      t1, 0(A)
st      t2, 2(A)
winning:
```

The ROP Method

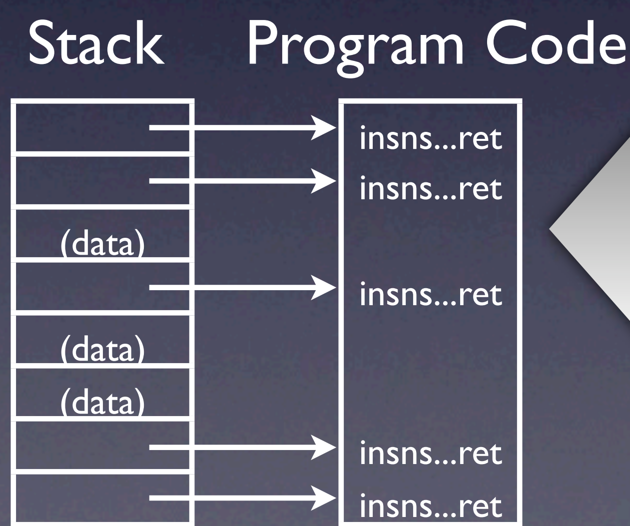
High-level specification

```
if arnold ≤ washington:
    amount = (washington - arnold)/2 + 1
    arnold = arnold + amount
    washington = washington - amount
```

Pseudo-assembly

```
ld      t1, 0(A)
ld      t2, 2(A)
slt     t3, t2, t1
btr     t3, winning
sub     amt, t2, t1
srl     amt, amt, 1
inc     amt
sub     t2, t2, amt
add     t1, t1, amt
st      t1, 0(A)
st      t2, 2(A)
winning:
```

Gadgets



The Usual Method

- Sequence of instructions: %eip
- Execute instruction, update %eip
- Control flow by changing %eip

```
%eip →  movl    ..., %edx
        movl    ..., %ecx
        compl   %ecx, %edx
        jg      winning
        movl    %ecx, %eax
        subl    %edx, %eax
        shrl    %eax
        incl    %eax
        addl    %eax, %edx
        movl    %edx, ...
        subl    %eax, %ecx
        movl    %ecx, ...
        winning:
```

The Usual Method

- Sequence of instructions: %eip
- Execute instruction, update %eip
- Control flow by changing %eip

```
         movl   ..., %edx
         movl   ..., %ecx
         compl  %ecx, %edx
%eip →   jg     winning
         movl   %ecx, %eax
         subl   %edx, %eax
         shrl   %eax
         incl   %eax
         addl   %eax, %edx
         movl   %edx, ...
         subl   %eax, %ecx
         movl   %ecx, ...
         winning:
```

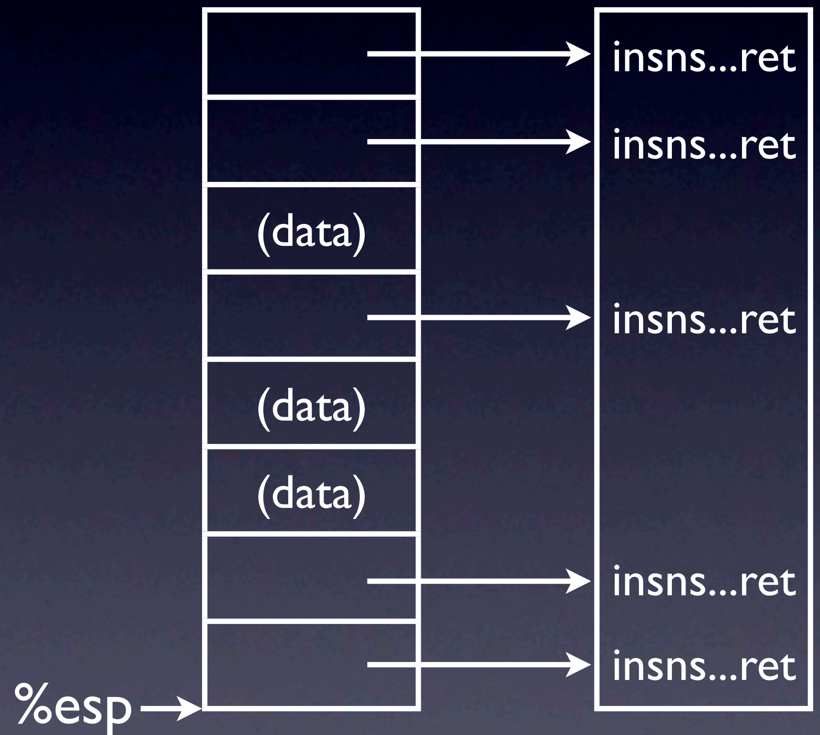
The Usual Method

- Sequence of instructions: %eip
- Execute instruction, update %eip
- Control flow by changing %eip

```
        movl    ..., %edx
        movl    ..., %ecx
        compl   %ecx, %edx
        jg      winning
        movl    %ecx, %eax
        subl    %edx, %eax
        shrl    %eax
        incl    %eax
        addl    %eax, %edx
        movl    %edx, ...
        subl    %eax, %ecx
        movl    %ecx, ...
%eip →  winning:
```

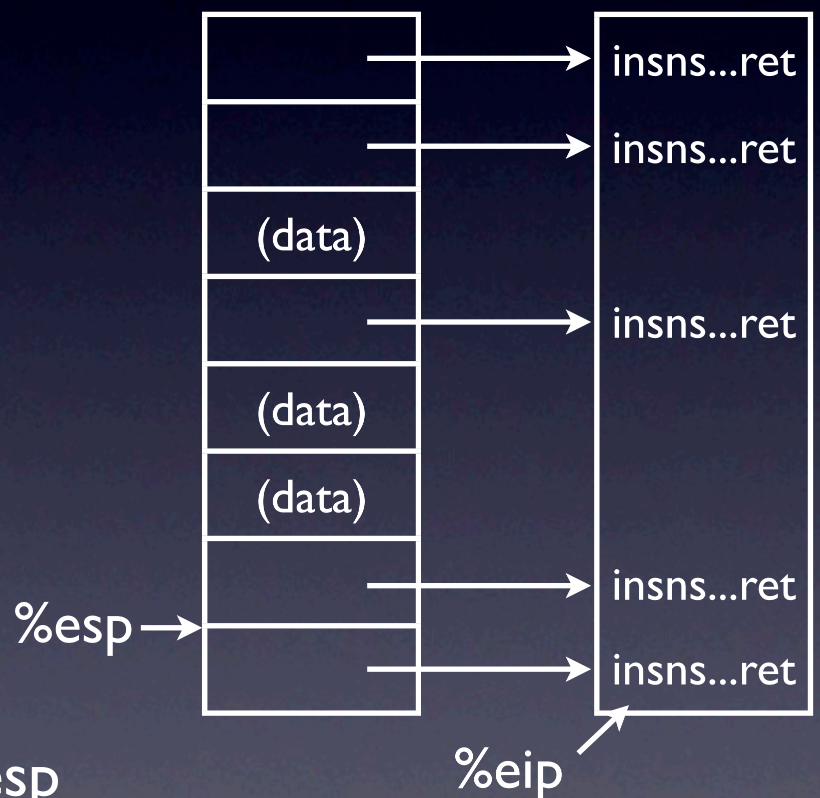
The ROP Method

- Sequence of Gadgets: %esp
  - Pointers to instructions
  - Data
- Execute Gadget
  - ret increments %esp
- Control flow by changing %esp



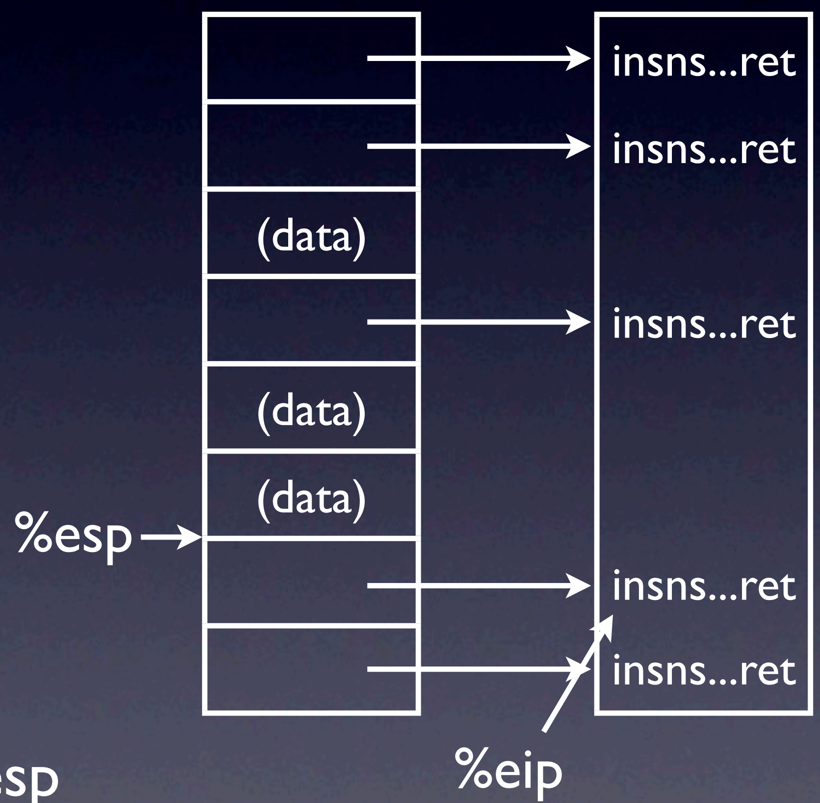
The ROP Method

- Sequence of Gadgets: %esp
  - Pointers to instructions
  - Data
- Execute Gadget
  - ret increments %esp
- Control flow by changing %esp



The ROP Method

- Sequence of Gadgets: %esp
  - Pointers to instructions
  - Data
- Execute Gadget
  - ret increments %esp
- Control flow by changing %esp



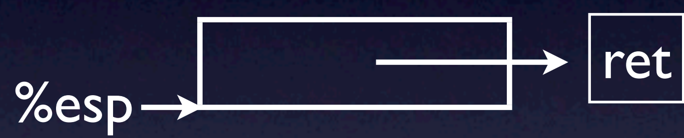
ROP Example 1: No-op

Usual

%eip → nop

- Just advances %eip

ROP



- Just advances %esp
- Pointer to ret instruction

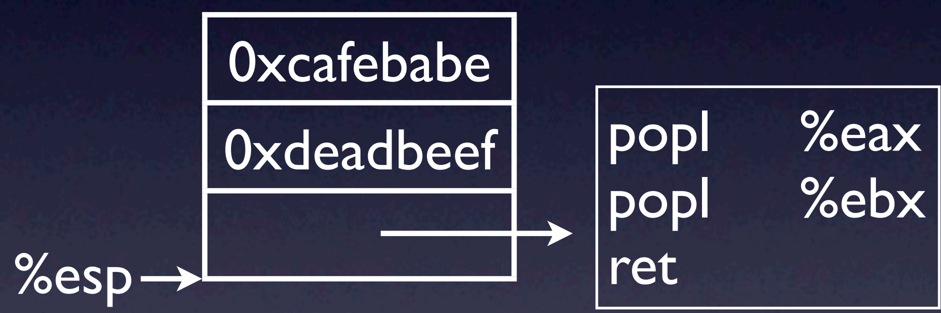
ROP Example 2: Immediate Constants

Usual

```
%eip → movl $0xdeadbeef, %eax
       movl $0xcafebabe, %ebx
```

- Set %eax to 0xdeadbeef
- Set %ebx to 0xcafebabe

ROP



- Put constants on stack
- Pop them into registers

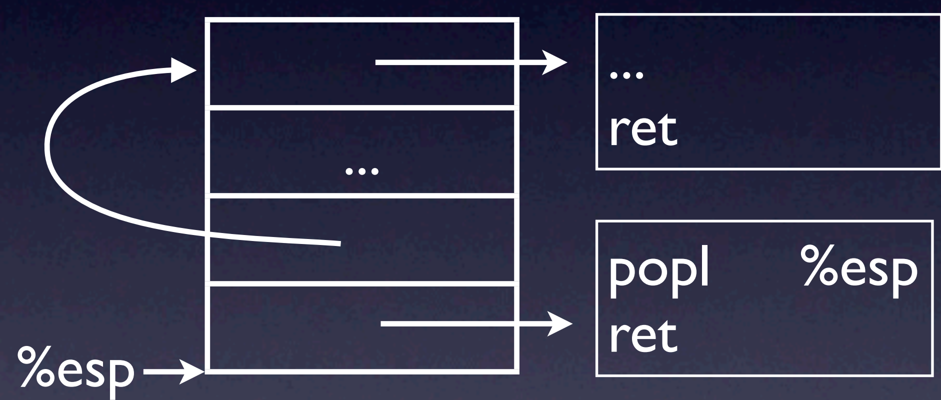
ROP Example 3: Control Flow

Usual

%eip → jmp +16

- Update %eip

ROP



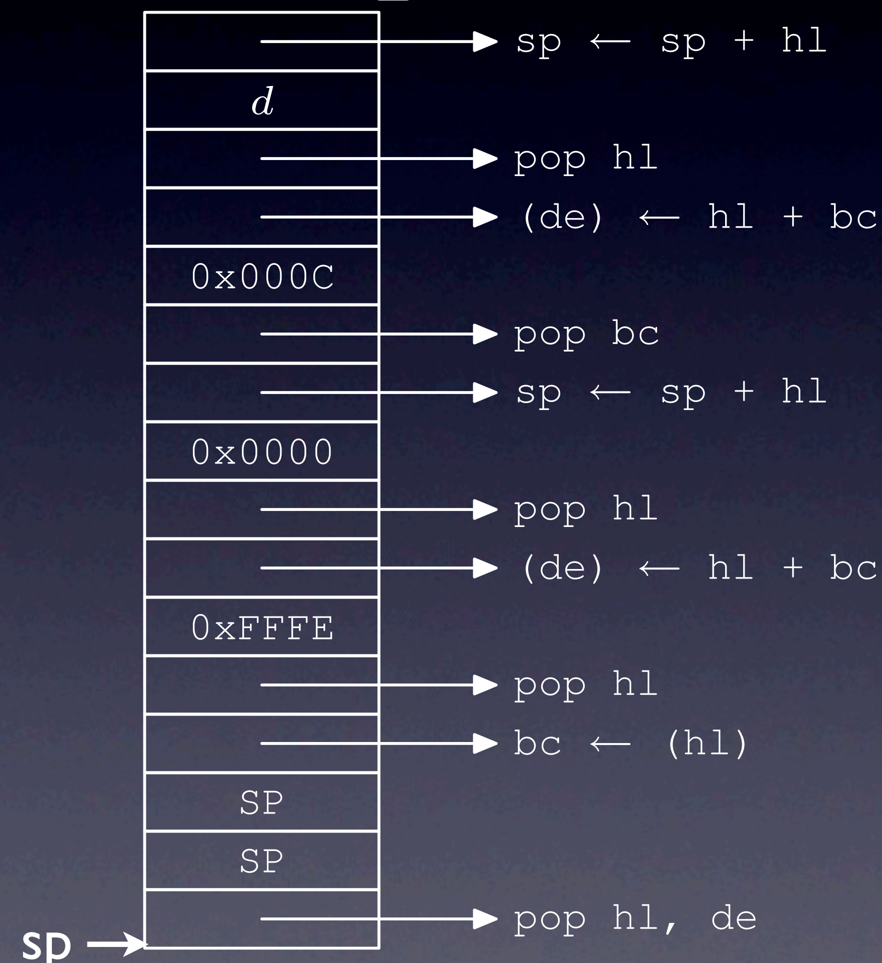
- Update %esp
- Conditional branch possible

ROP Wrap-Up

- Use stack for program (%esp vs. %eip)
- Gadgets
  - Multiple instruction sequences & data
  - Chained together by ret
- Turing-complete
- No code injection!

ROP On The AVC Advantage

- Extended ROP to Z80
- 16 kB instruction corpus
- Turing-complete gadget set
    - Some automation



Challenges Overcome

1. Reverse-engineered hardware and software
2. Found an exploitable bug in the code
3. Defeated code-injection defense using return-oriented programming

Thank you