Where's the FEEB?
The Effectiveness of Instruction Set Randomization

Ana Nora Sovarel, David Evans, Nathanael Paul
University of Virginia
USENIX 2005

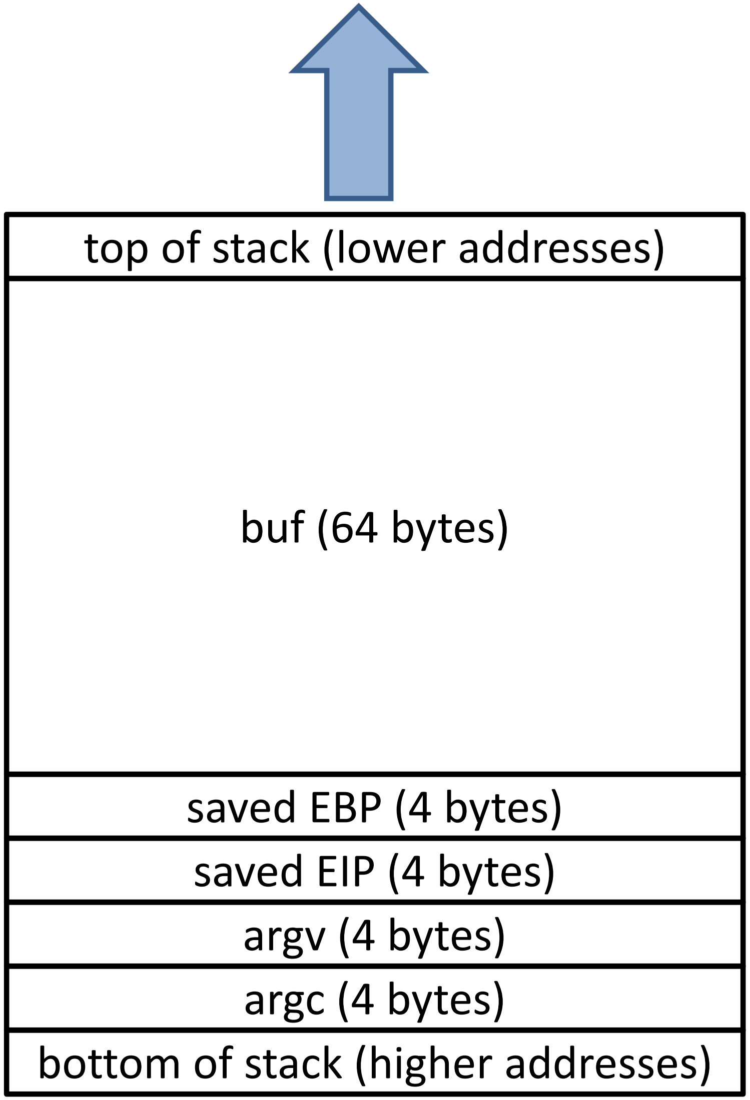
Code-Injection Attacks

- Inject malicious executable code (payload) into victim process
  - e.g., via attacker-supplied input
- Convince victim process to execute payload
  - e.g., leverage buffer overrun to overwrite return address
- Attacker acquires complete control of process and all its privileges

Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

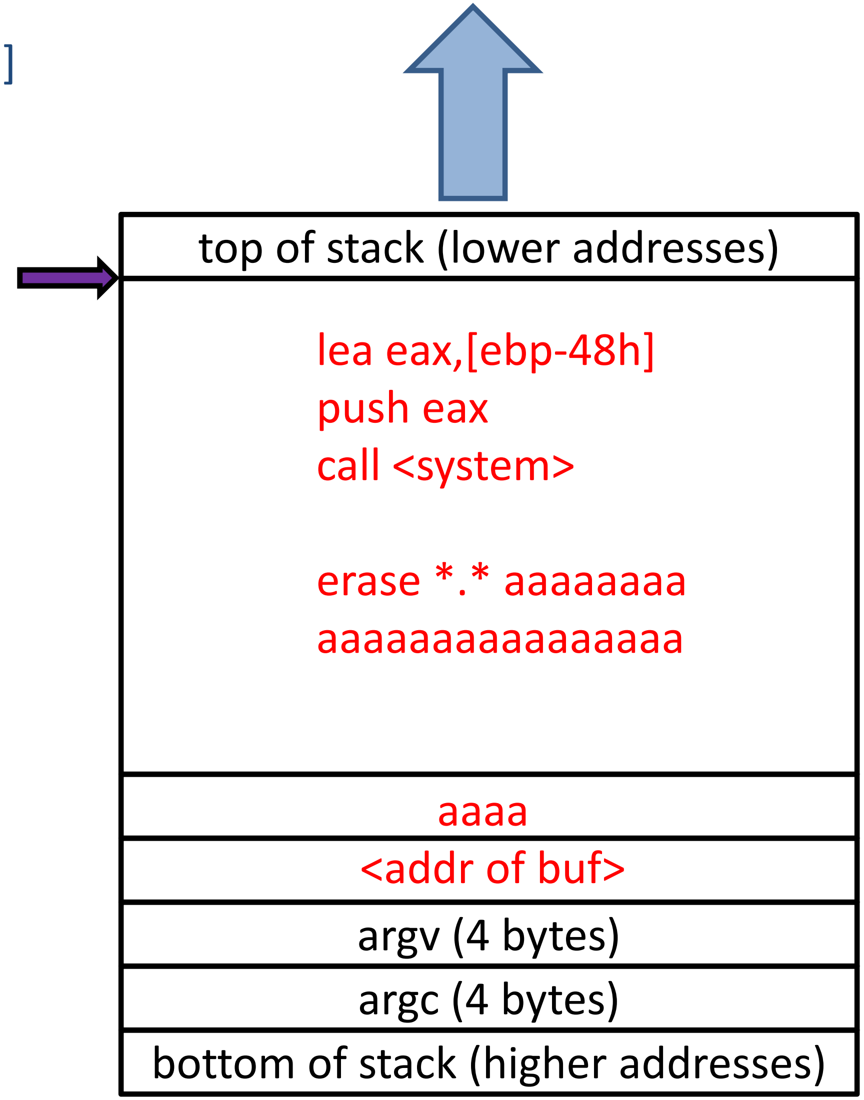
```
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    …
    return;
}
```

top of stack (lower addresses)
lea eax,[ebp-48h] push eax call <system> erase *.* aaaaaaaa aaaaaaaaaaaaaaaa
aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)

Code-injection Example

```
8D 45 B8             lea eax,[ebp-48h]
50                   push eax
FF 15 BC 82 2F 01    call <system>
65 72 61 73 65 20    .data “erase ”
2A 2E 2A 20          .data “*.* ”
61 (x24)             .data “aaaaa…”
61 61 61 61          .data “aaaa”
30 FB 1F 00          <addr of buf>
```

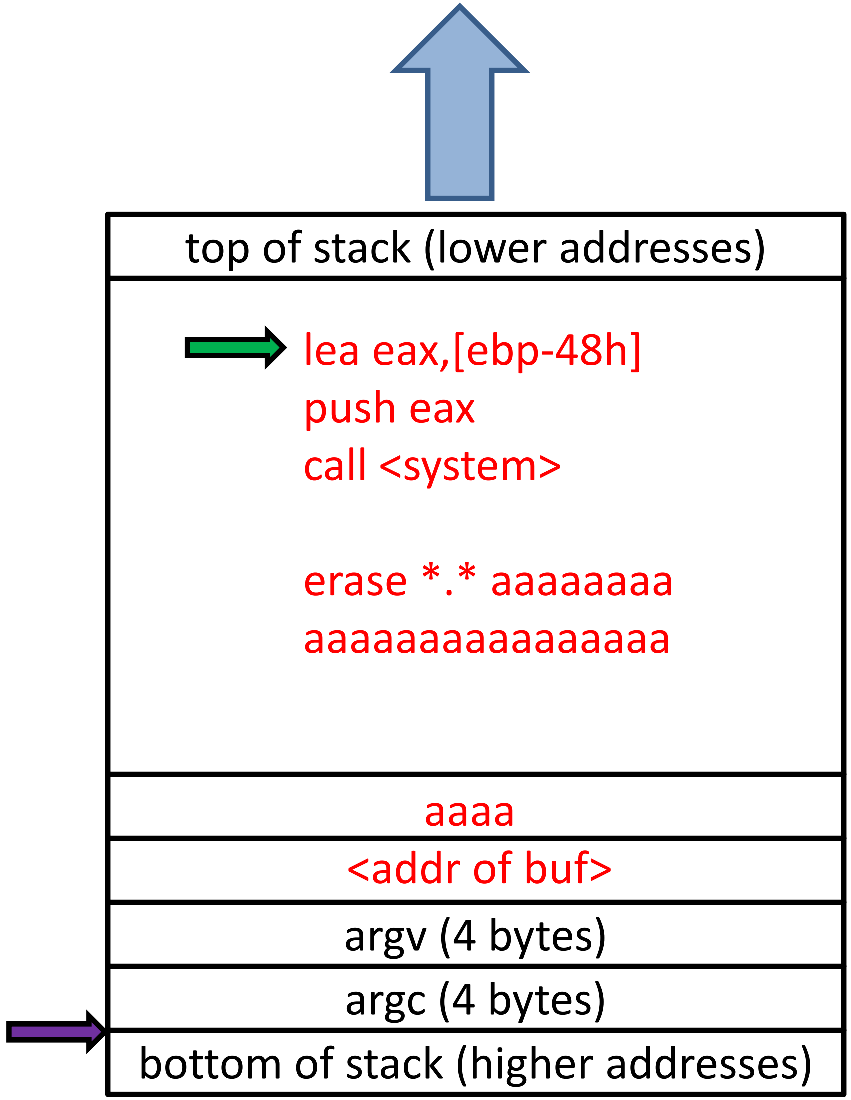
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

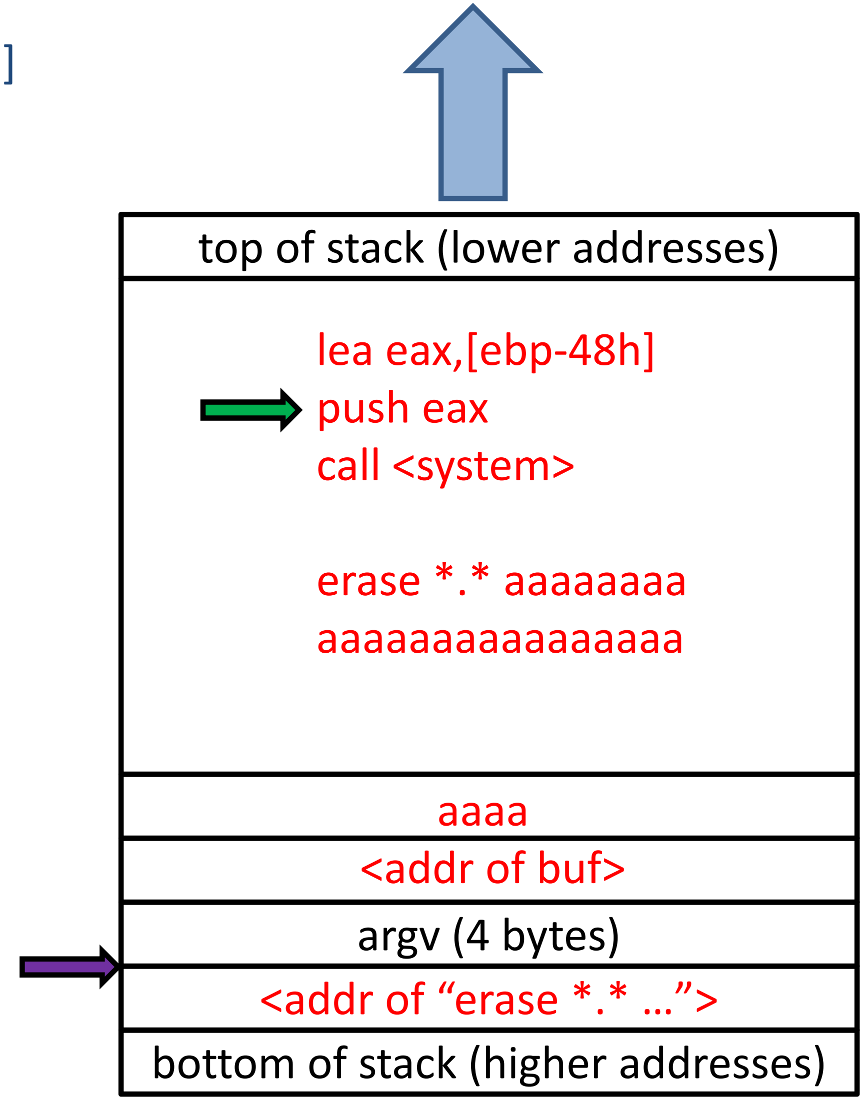
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

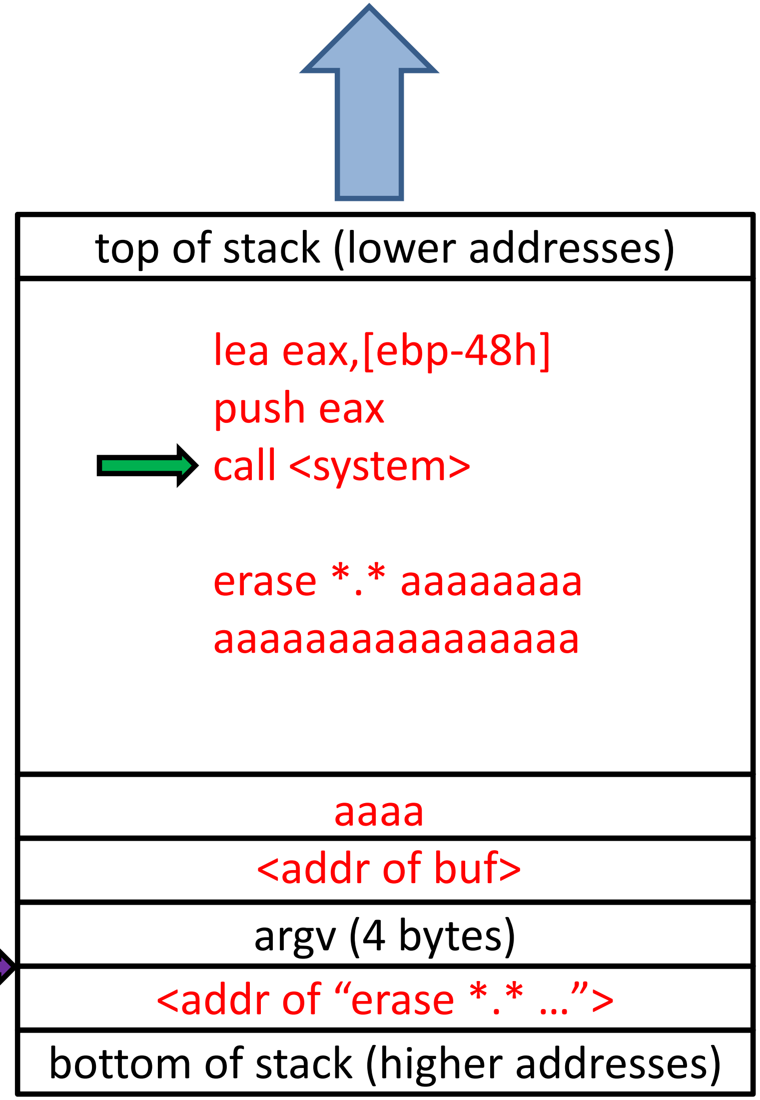
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

```
8D 45 B8             lea eax,[ebp-48h]
50                   push eax
FF 15 BC 82 2F 01    call <system>
65 72 61 73 65 20    .data "erase "
2A 2E 2A 20          .data "*.* "
61 (x24)             .data "aaaaa…"
61 61 61 61          .data "aaaa"
30 FB 1F 00          <addr of buf>
```

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    …
    return;
}
```



Instruction Set Randomization

- Idea: Randomize the opcode encodings
  - Secure CPU has privileged 8-bit KEY register
  - CPU xor’s each fetched instruction byte with KEY before interpreting (decrypting it)
  - OS xor’s entire program text with KEY at load-time (encrypting it in memory)
- Better implementation:
  - Key is a length-n byte sequence
  - CPU xor’s code at address i with KEY[i mod n]

Code-injection Example

8D 45 B8	<random instructions>
50	<random instructions>
FF 15 BC 82 2F 01	<random instructions>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```

top of stack (lower addresses)
<random instructions> <random instructions> <random instructions> erase *.* aaaaaaaa aaaaaaaaaaaaaaaa
aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)

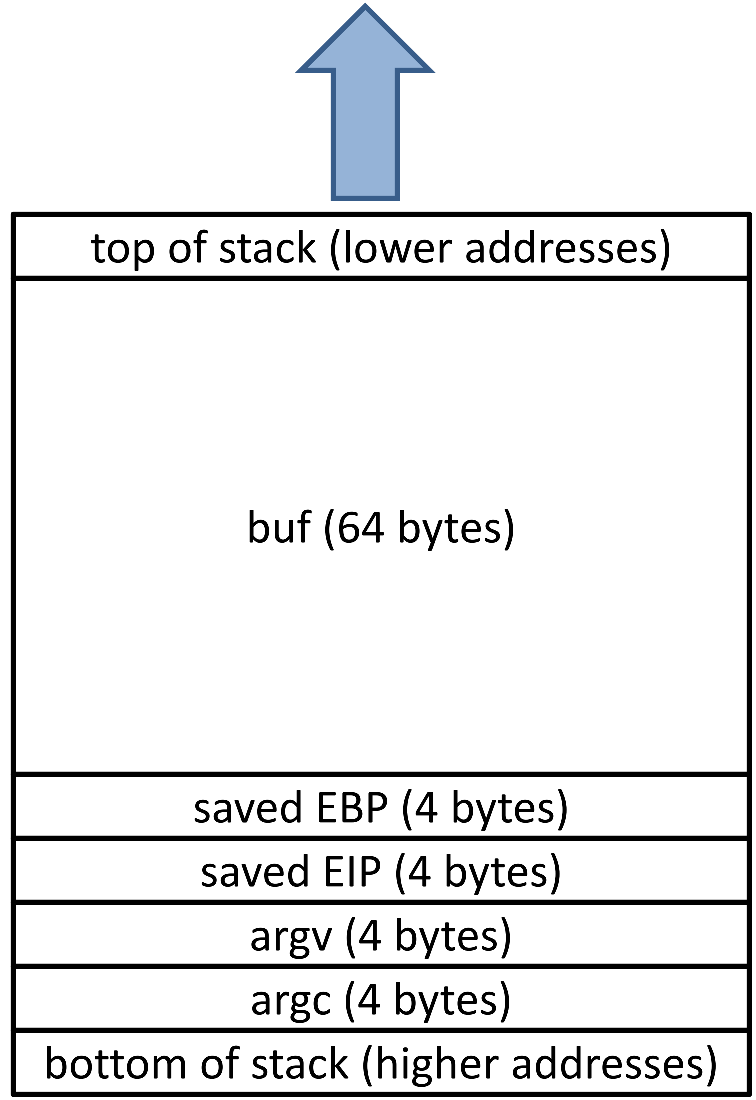
Attacking ISR

- Goal: Discover the KEY (or at least some of it)
- Four-phase attack:
  - Phase 1: discover 1 or 2 bytes of the KEY
  - Phase 2: discover 4 bytes of the KEY
  - Phase 3: discover 100 bytes of the KEY
  - Phase 4: inject full-sized malicious payloads

Phase 1: Return-attack

XX	ret?
61 (x63)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>
61 (x8)	.data “aaaaaaaa”
03 14 DF 01	<original return addr>

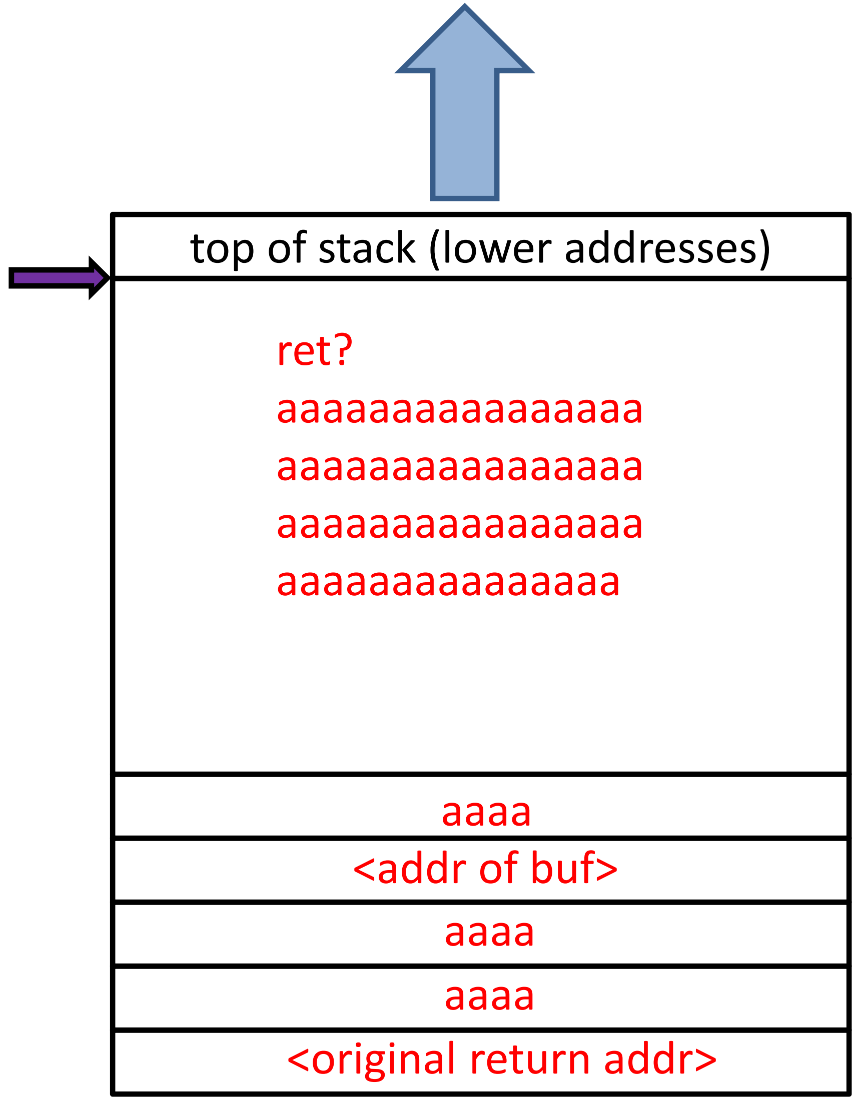
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Return-attack

XX	ret?
61 (x63)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>
61 (x8)	.data “aaaaaaaa”
03 14 DF 01	<original return addr>

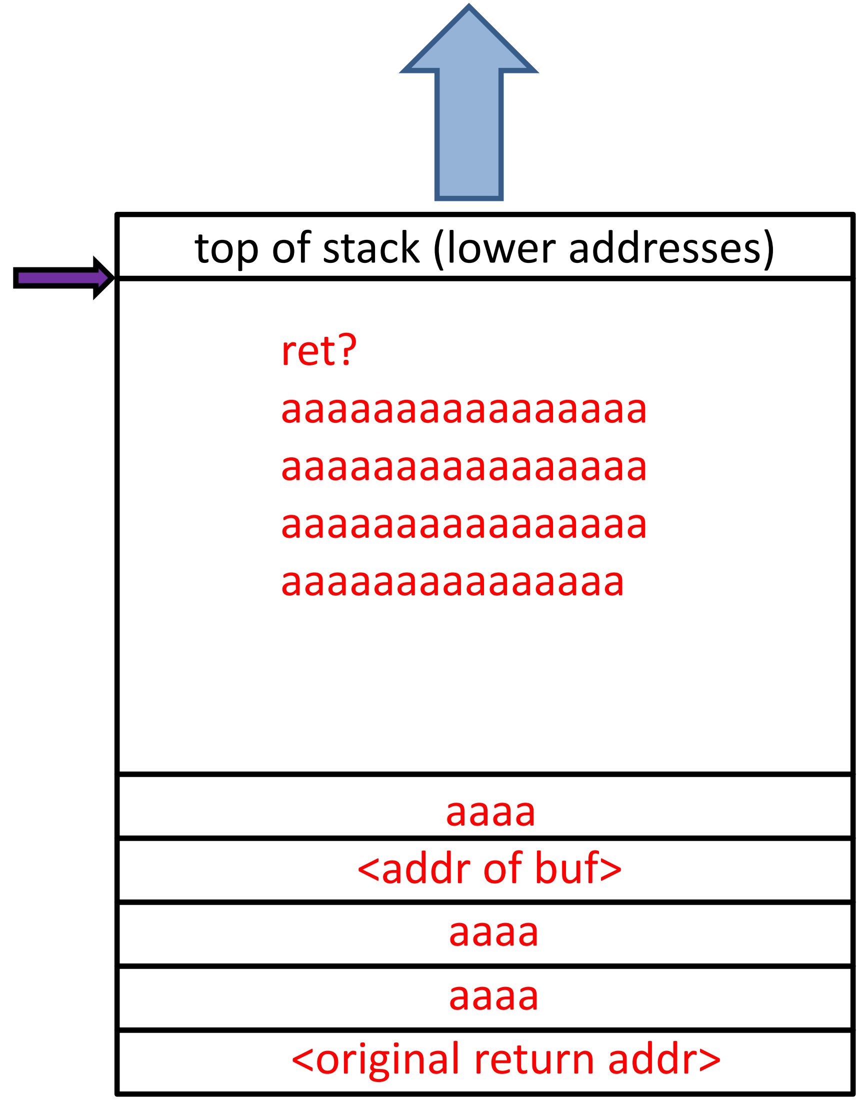
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Return-attack

XX	ret?
61 (x63)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>
61 (x8)	.data “aaaaaaaa”
03 14 DF 01	<original return addr>

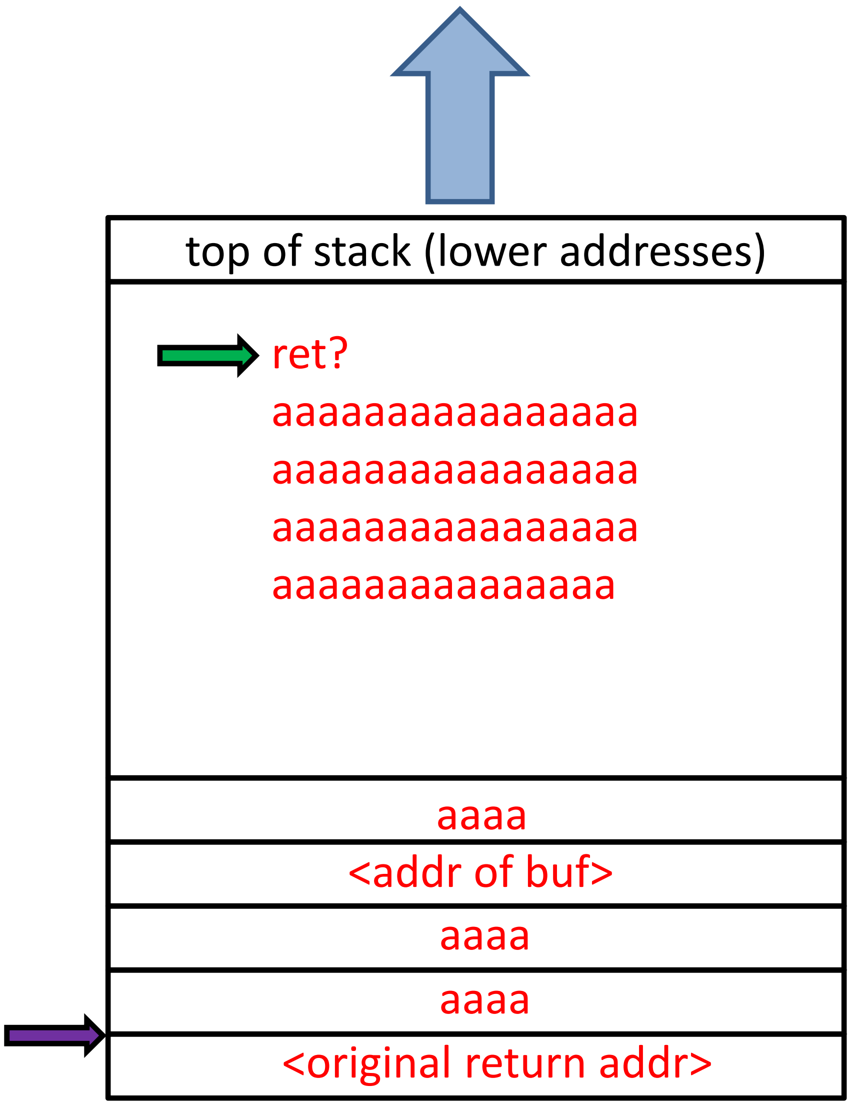
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Return-attack

XX	ret?
61 (x63)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>
61 (x8)	.data “aaaaaaaa”
03 14 DF 01	<original return addr>

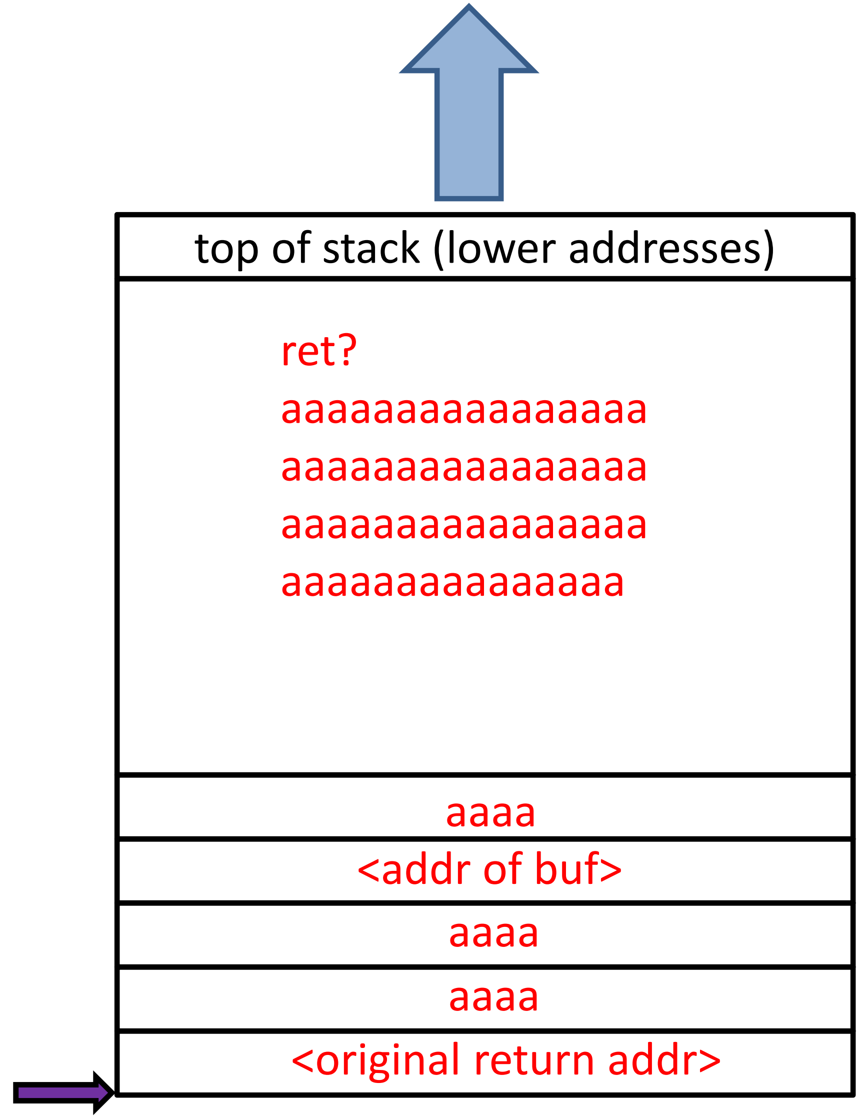
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Return-attack

XX	ret?
61 (x63)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>
61 (x8)	.data “aaaaaaaa”
03 14 DF 01	<original return addr>

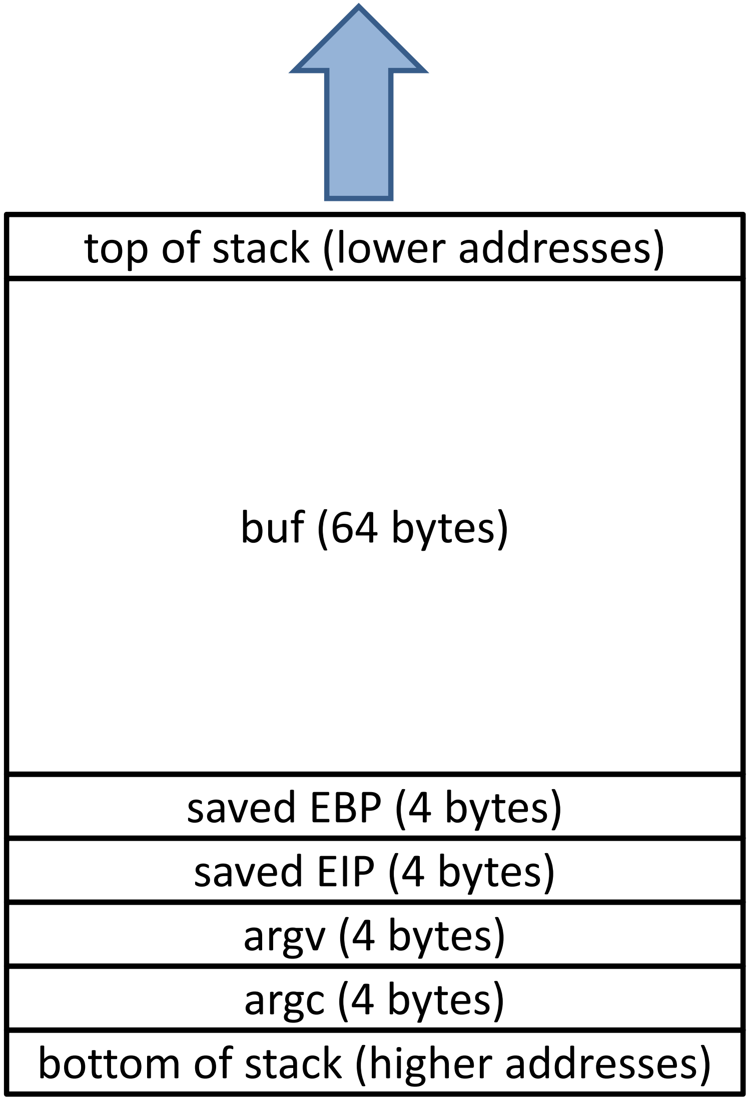
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Jump-attack

XX XX	loop: jump loop?
61 (x62)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

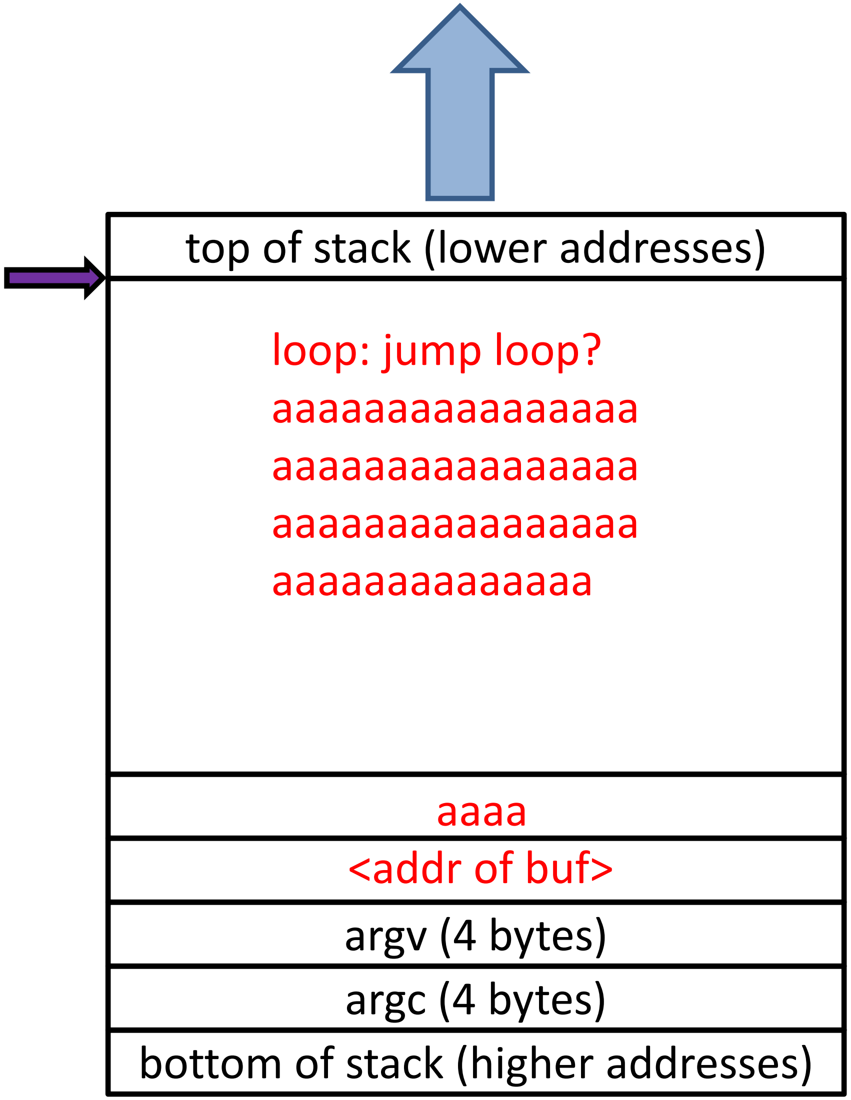
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Jump-attack

XX XX	loop: jump loop?
61 (x62)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

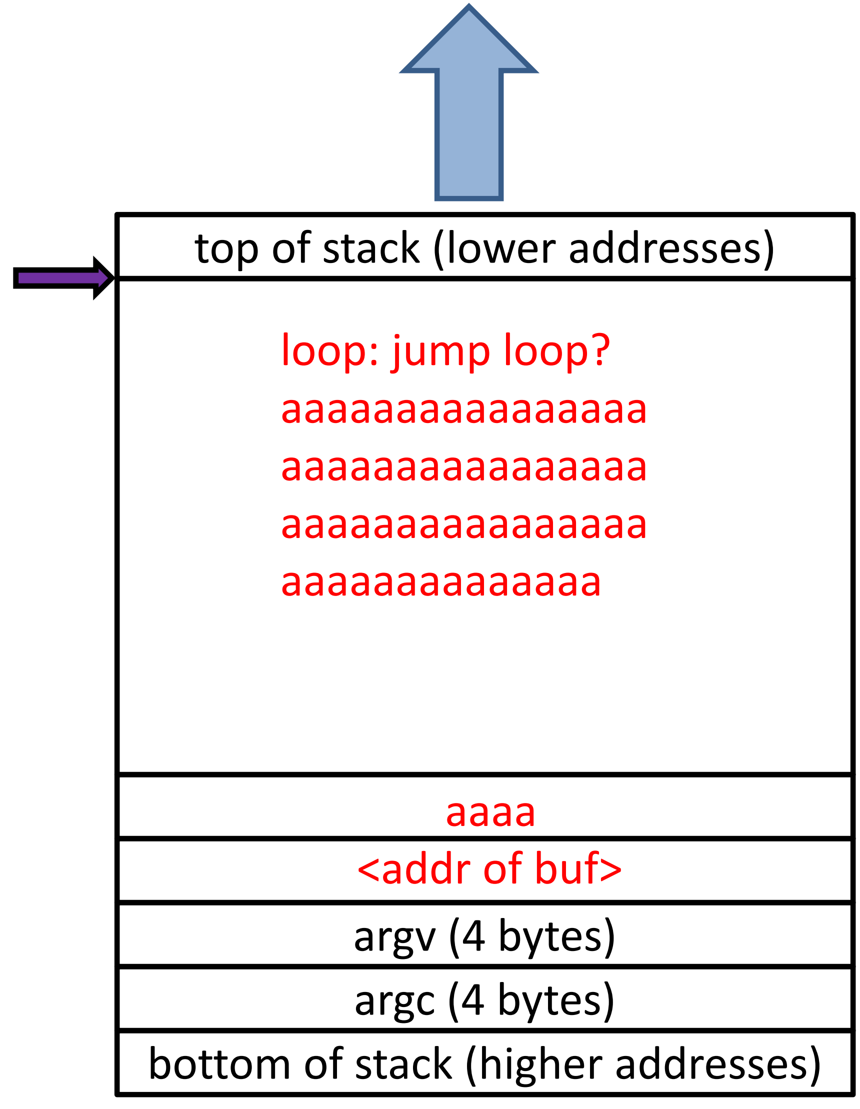
```
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    …
    return 0;
}
```



Phase 1: Jump-attack

XX XX	loop: jump loop?
61 (x62)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

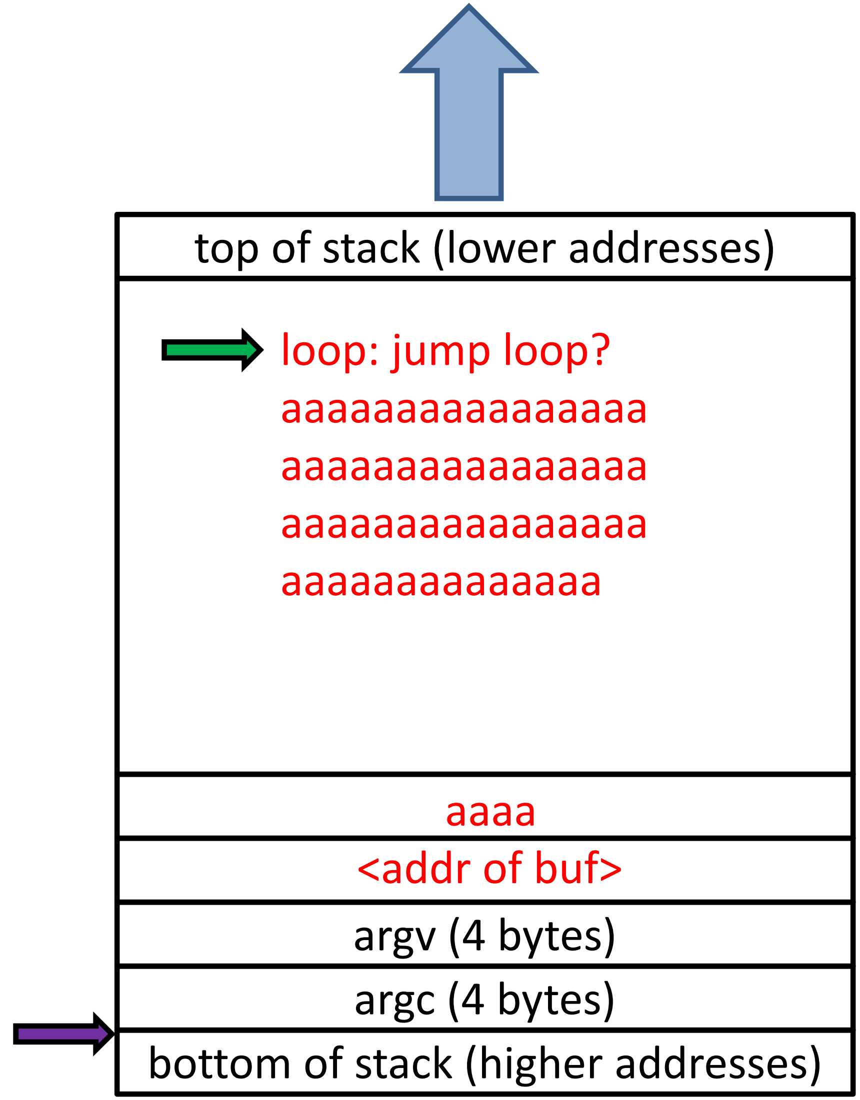
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 1: Jump-attack

XX XX	loop: jump loop?
61 (x62)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 2: Jump-attack

90	nop
90	nop
XX XX	loop: jump loop?
61 (x60)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```

top of stack (lower addresses)
nop nop loop: jump loop? aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaa
aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)

Phase 3: Jump-attack

EB 03 14 DF XX jump <original ret addr?>
61 (x59) .data “aaaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>

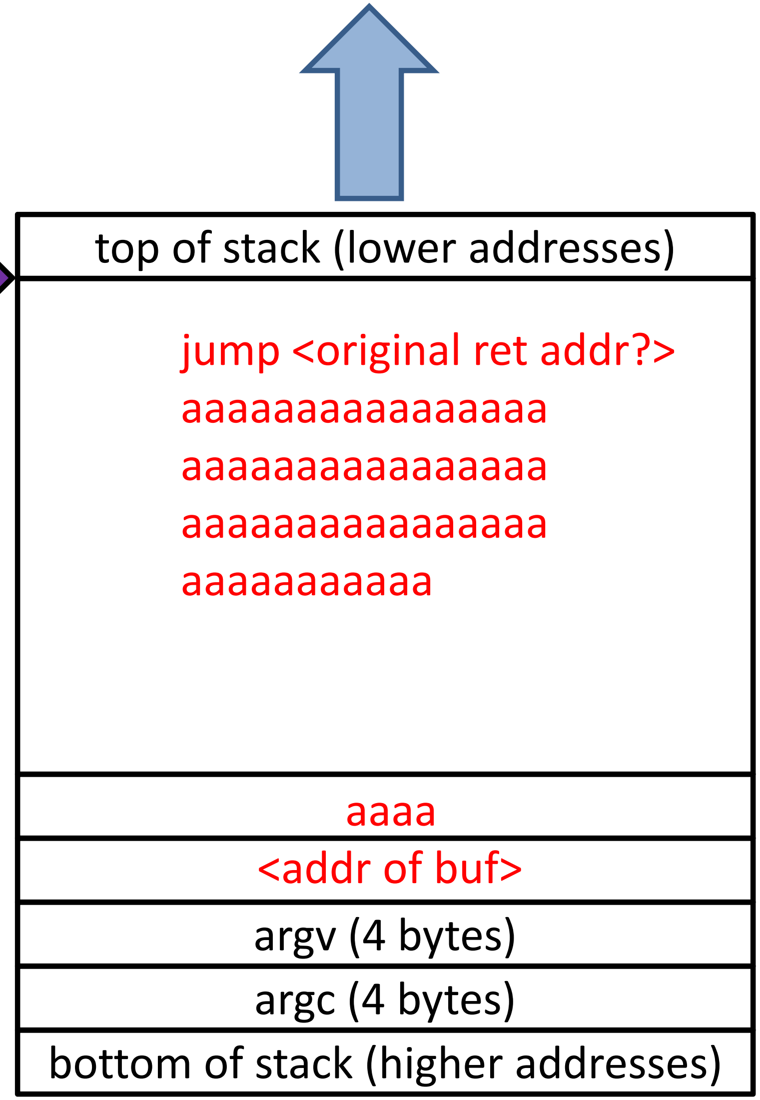
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```

top of stack (lower addresses)
jump <original ret addr?> aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaa
aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)

Phase 3: Jump-attack

EB 03 14 DF XX	jump <original ret addr?>
61 (x59)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 10	<addr of buf>

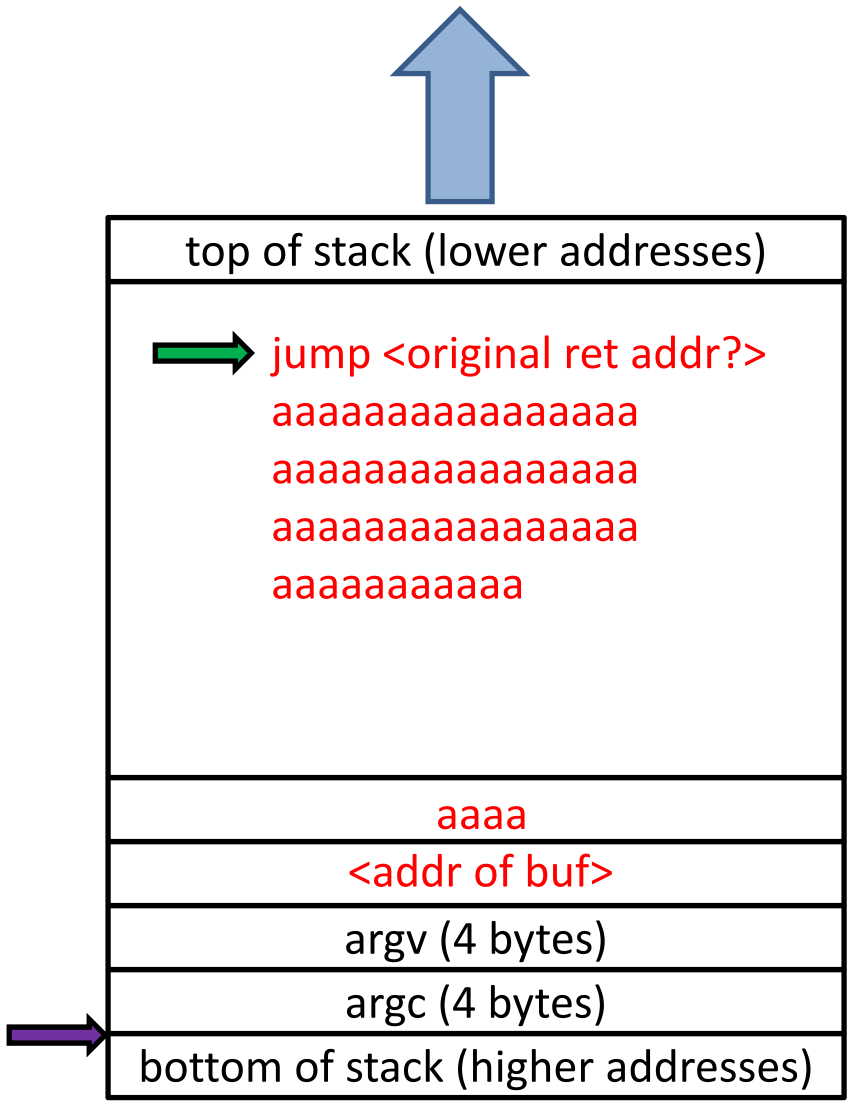
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 3: Jump-attack

EB 03 14 DF XX jump <original ret addr?>
61 (x59) .data “aaaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>

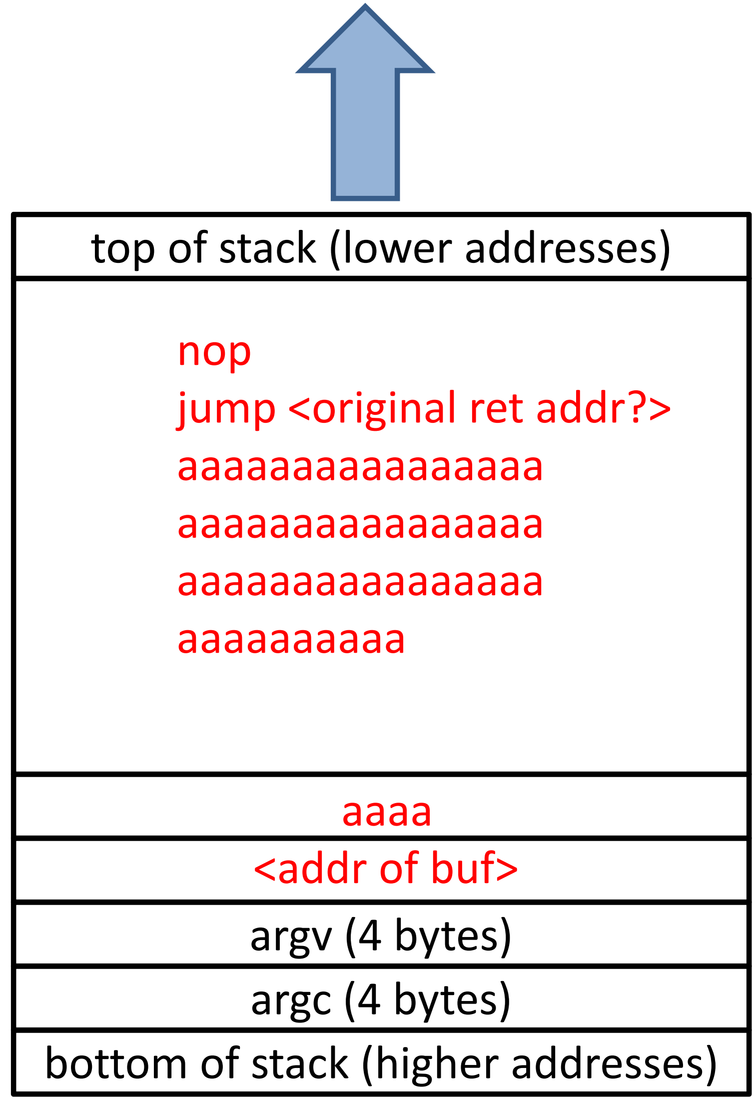
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phase 3: Jump-attack

```
90               nop
EB 03 14 DF XX   jump <original ret addr?>
61 (x58)         .data "aaaaa…"
61 61 61 61      .data "aaaa"
30 FB 1F 10      <addr of buf>
```

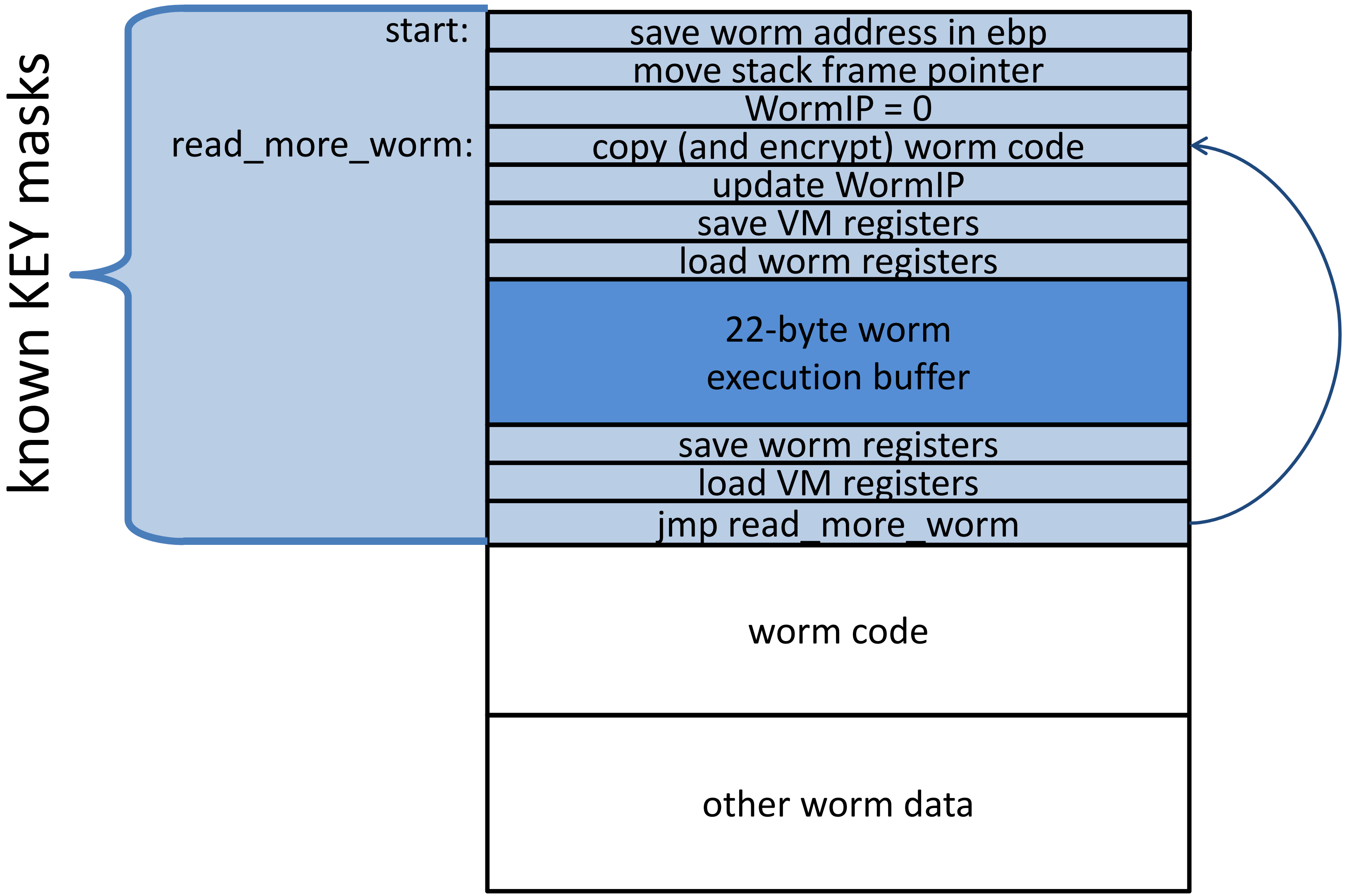
```
int main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return 0;
}
```



Phases 4: Full-size Payloads

- Learn ~100 bytes of KEY using Phases 1-3
- Goal: Construct a payload such that…
  - execution of payload never steps IP outside the 100-byte window of known KEY's
  - payload can be much larger than 100 bytes
- Solution: inject a virtual machine!
  - main engine of VM confined to 100-byte window
  - VM copies small chunks of payload into window
  - copying process encrypts using known KEY bytes
  - chunk returns back to main engine when next chunk required

Phase 4: MicroVM



Technical Issues

- False positives
  - probabilistic analysis and mitigation strategies
  - (see Section 3 of paper)
- Payloads that contain null bytes
  - compute them dynamically (e.g., “xor eax,eax” instead of “mov eax,0x00000000”)
- ISR’s that re-randomize after crashes
  - only an issue when children crash parent process
  - questionable ISR design choice
  - no easy workaround suggested, though…

Experimental Results

- Jump-attack
  - cracked 100-byte key in ~6 min. average
  - success rate: 95-100%
  - ~9 infinite loops on average

Improving ISR

- Larger instruction encodings
  - RISC: all instructions 32-bits long
- Better encryption
  - AES instead of XOR
  - (too expensive to be practical)
- Non-uniform remapping of instructions
  - introduce P[255], a random permutation of 0..255
  - to decrypt byte b at address i, compute (P[b] xor KEY[i])
  - encryption uses inverse P table
  - Why does this defeat the attack?

Selected References

Sovarel, Evans, and Paul. **Where's the FEEB? The Effectiveness of Instruction Set Randomization**. *Proc. 14th USENIX Security Symposium*, 2005.

Shacham, Page, Pfaff, Goh, Modagugu, and Boneh. **On the Effectiveness of Address-Space Randomization**. *Proc. 11th ACM Conf. on Comp. and Comm. Security*, 2004.