von Neumann and the Current Computer Security Landscape

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu
This talk

• Overview of the von Neumann computer architecture
• Security implications
  • software vulnerabilities
  • limitations in detecting malware
  • defenses that play on the architecture
John von Neumann

- Mathematician, instrumental in the developing
- quantum mechanics
- cellular automata
- economics & game theory
- nuclear physics
- computer architecture
von Neumann architecture

- Unified memory for instructions and data
- Contrast: Harvard architecture
- Specified in tech report on EDVAC in 1945
  - Similar ideas floating previously
- Simplicity led to wide acceptance
  - Practically all modern computers based on this architecture
Corollary

- Code and data look “the same”
- is \texttt{0x90} data or an x86 instruction?
- We must somehow differentiate between code and data
- Program and/or OS must know
- debugging is easy (or easier)
Corollary (2)

• Code can be treated as data
  • self-modifying code
  • dynamic code generation
  • debugging

• Code is treated as data
  • copy a program vs. run a program
Performance implications

- Performance bottleneck due to shared memory bus
- "von Neumann bottleneck"
- led to the development of caches, branch prediction, etc.
- For many years, this was the main issue
Implications for reliability

- Mistaking data and instructions leads to undefined behavior
- CPU will try to execute data as instructions
  - for random data, this will cause exception (memory, opcode, etc.)
- code-as-data can be modified
- RO code pages to avoid mistakes
Implications for security

• What if random data is not random?
  • data is/contains code
  • code can be written by attacker

• Program will end up executing foreign code that will do the attacker’s work

• Privileges of program/user or of program source
Security problems

- Viruses
- Detection of malware
- Code injection attacks
  - buffer overflows
  - SQL injection
  - Cross-site Scripting (XSS)
Viruses

• Self-propagating code

• First “large scale” outbreaks in 1981, for MS-DOS
  • infected executable files (.exe, .com)
  • treated code (programs) as data
  • modified binaries to insert themselves
Virus-infected file

Program

Virus

Program
Virus detection

• Anti-virus programs typically look for “signatures” (byte strings) of known viruses
  • prior to program execution, after download, incoming email attachments, etc.

• Attackers’ response: polymorphism
Polymorphism

- Two-part viruses
  - small first part (decoder) decrypts second part
  - second part contains main attack payload
- Signatures on second part are difficult/impossible
- Small decoder means signatures are likely to have false positives
Polymorphism

- Increasing use in all kinds of malware
  - viruses, worms, trojans, etc.
  - self-extracting “packers”
- Attackers can create large numbers of decoders
Code injection attacks

- Programs may be tricked into treating input data as code
  - data received over the network or otherwise supplied by an untrusted user
  - exploit weaknesses in input validation to overwrite control information
Buffer overflow attacks

- Specific instance of code injection in C/C++ (and similar languages)
- Function return address in function frame is overwritten with attacker-controlled data
- Same data contains attack code

```c
int caller(char *str) {
    char buf[100];
    strcpy(buf, str);
}
```
Buffer overflow attacks

• Specific instance of code injection in C/C++ (and similar languages)

• function return address in function frame is overwritten with attacker-controlled data

• same data contains attack code

```c
char *str;
char buf[100];
caller(char *str) {
    char buf[100];
    strcpy(buf, str);
}
```
Note on buffer overflows

• There are many different variants
• not all inject code
  • e.g., “return-into-libc” attacks
• some compromise control data in other ways
• All end up subverting the control flow of the program to meet attacker’s goals
Real problem

- Many vulnerabilities discovered daily on commercial and open-source software
- enable remote compromise
- typically also confer superuser privileges to attacker
- enabling technology for fast-spreading worms
Buffer overflow prevalence

Source: NIST
Buffer overflow prevalence

![Bar chart showing the prevalence of buffer overflows from 2001 to 2008. The chart indicates a significant increase in 2008.]
Code injection prevalence
Code injection prevalence

![Bar chart showing code injection prevalence over years]

- % of Total Vulnerabilities Meeting Specified Limitations
- Prevalence peaks in 2008 and 2007, with significant reductions in subsequent years.
Defenses?

- Network and A/V-style defenses seem problematic (re: polymorphism)
- Drastic change (e.g., safe languages) is slow and difficult
- Move closer to the host/software
  - detect symptoms of attack
  - slow and difficult to scale defenses
- Model legitimate inputs rather than detect anomalous inputs
- Open area(s) of research and practice
Some interesting defenses

- Hardware support *(NX bit)*
- Secrecy-based separation
  - Instruction-Set Randomization
  - Address Space Obfuscation
Randomization
Randomization

0x08048262 <foobar+122>: add $0x10,%esp
0x08048265 <foobar+125>: mov 0x8(%ebp),%eax
0x08048268 <foobar+128>: mov 0x8(%ebp),%edx
0x0804826b <foobar+131>: mov (%edx),%edx
0x0804826d <foobar+133>: add $0xa,%edx
0x08048270 <foobar+136>: mov %edx,(%eax)
Randomization

0x08048262 <foobar+122>:   add   $0x10,%esp
0x08048265 <foobar+125>:   mov   0x8(%ebp),%eax
0x08048268 <foobar+128>:   mov   0x8(%ebp),%edx
0x0804826b <foobar+131>:   mov   (%edx),%edx
0x0804826d <foobar+133>:   add   $0xa,%edx
0x08048270 <foobar+136>:   mov   %edx,(%eax)

code_slice XOR 0xA7 produces:
Randomization

```
0x08048262 <foobar+122>:    add $0x10,%esp
0x08048265 <foobar+125>:    mov 0x8(%ebp),%eax
0x08048268 <foobar+128>:    mov 0x8(%ebp),%edx
0x0804826b <foobar+131>:    mov (%edx),%edx
0x0804826d <foobar+133>:    mov %edx,(%eax)
0x08048262 <foobar+122>:    and $0x63,%al
0x08048264 <foobar+124>:    mov $0x2c,%bh
0x08048266 <foobar+126>:    loop 0x8048217 <foobar+47>
0x08048268 <foobar+128>:    sub $0xf2,%al
0x08048268 <foobar+128>:    sub $0xf2,%al
0x0804826a <foobar+130>:    scas %es:(%edi),%eax
0x0804826b <foobar+131>:    sub $0xb5,%al
0x0804826d <foobar+133>:    and $0x65,%al
0x0804826f <foobar+135>:    lods %ds:(%esi),%eax
0x08048270 <foobar+136>:    cs
```

code_slice XOR 0xA7 produces:

```
0x08048262 <foobar+122>:    and $0x63,%al
0x08048264 <foobar+124>:    mov $0x2c,%bh
0x08048266 <foobar+126>:    loop 0x8048217 <foobar+47>
0x08048268 <foobar+128>:    sub $0xf2,%al
0x0804826a <foobar+130>:    scas %es:(%edi),%eax
0x0804826b <foobar+131>:    sub $0xb5,%al
0x0804826d <foobar+133>:    and $0x65,%al
0x0804826f <foobar+135>:    lods %ds:(%esi),%eax
0x08048270 <foobar+136>:    cs
```
SQL injection

- Code injection attacks are not limited to binaries

```
SELECT * from items
where customer_name='$USERNAME';
```
• Code injection attacks are not limited to binaries

```
SELECT * from items
where customer_name='$USERNAME';
```
SQL injection

- Code injection attacks are not limited to binaries

```sql
SELECT * from items
where customer_name='$USERNAME';
```

```
SELECT * from items
where customer_name='angelos'
```
**SQL injection**

- Code injection attacks are not limited to binaries

```sql
SELECT * from items
where customer_name='$USERNAME';
```

SELECT * from items
where customer_name='angelos';
SQL injection

- Code injection attacks are not limited to binaries

```sql
SELECT * from items
where customer_name='$USERNAME';
```
• Code injection attacks are not limited to binaries

SELECT * from items
where customer_name='$USERNAME';

username=' or username is not null
or username = '
SQL injection

• Code injection attacks are not limited to binaries

SELECT * from items
where customer_name='$USERNAME';

User ← SQL → Database Server

username=' or username is not null
or username = '

SELECT * from items
where customer_name='' or
username is not null or
username = '';
Significance

• Another instance of mixing data and code
• not direct result of von Neumann architecture
• result of decades of mentally ignoring the difference between code and data
SQL injection prevalence
SQL injection prevalence

![Bar chart showing SQL injection prevalence over years]

- 2009: 16
- 2008: 2
- 2007: 1
- 2006: 0
- 2005: 0
- 2004: 0
- 2003: 0
- 2002: 0

% of Total Vulnerabilities Meeting Specified Limitations
Command injection

- The problem does not end with SQL injection
- any interpreted language that receives untrusted input is susceptible
- PHP, Perl, shell script, ...
Taint tracking

- Modify runtime environment (e.g., Perl interpreter) to track use of data from untrusted sources.
- Alert/stop if such data is used in sensitive operations.
- Variant for use with binaries.
  - Use emulation or hardware support.
    - Very slow.
SQL randomization

• Apply randomization to SQL templates

• Parameterize all keywords and operators

```sql
select gender, avg(age)
from cs101.students
where dept = %d
group by gender
```

```sql
select123 gender, avg123 (age)
from123 cs101.students
where123 dept =123 %d
group123 by123 gender
```

• Use de-randomizing proxy between client application and DBMS
Cross-Site Scripting (XSS)

• Web-oriented class of vulnerabilities
• Bypasses browser security sandbox
  • convinces browser (and user) that source of program is different (trusted?) site
• Programs are typically Javascript
  • can be other active content
How does it work?

- Some servers will mirror input from the URL in the returned page
- Error pages, naive applications, etc.
- Browsers don’t know the provenance of data in a returned page
How does it work?

- Some servers will mirror input from the URL in the returned page
- Error pages, naive applications, etc.
- Browsers don’t know the provenance of data in a returned page
How does it work?

- Some servers will mirror input from the URL in the returned page
- Error pages, naive applications, etc.
- Browsers don’t know the provenance of data in a returned page
How does it work?

- Some servers will mirror input from the URL in the returned page
- error pages, naive applications, etc.
- Browsers don’t know the provenance of data in a returned page
XSS in operation

Trusted Web Server

Untrusted Web Server

Browser
XSS in operation

Trusted Web Server

Browser

Untrusted Web Server

Get page
XSS in operation

Trusted Web Server

Untrusted Web Server

Redirect with JS embedded in URL

Get page

Browser
XSS in operation

Trusted Web Server

Redirect with JS embedded in URL

Browser

Untrusted Web Server

Get page

JS mirrored to browser
Notes on XSS

• Injected JS appears to come from trusted website
  • may fool the user through direct interaction
    • e.g., fake login prompt
  • can access cookies, issue direct requests against the trusted website
    • particularly powerful if user does not log out
XSS prevalence

![Bar chart showing XSS prevalence from 2000 to 2007.](chart.png)
XSS prevalence
XSS defenses

• No good known defenses

• Current state of practice
  • fix server configurations
  • fix applications
  • do not allow JS or other active content(?) from unknown websites
The future?

- Continuing mixing of code and data
- Data serialization formats such as JSON
- “Rich” document formats
  - Office, PDF, etc.
- Increasing focus on browser
Conclusion

• Overview of a large and important class of software vulnerabilities
• widely exploited on a daily basis
• difficult to get it right
• programmer education is lacking
• Historical perspective on architectural choices and their impact on security 40+ years later

• How do we change things, given current course?