Secure Project

Błażej Pietrzak
blazej.pietrzak@cs.put.poznan.p

Agenda

- **Designing**
  - Security principles
  - STRIDE Threat Model
  - ACL Lists
- **Programming**
  - Buffer Overrun
  - Cryptographic Foibles
  - Storing Secrets
  - Canonical Representation Issues

Agenda – cont.

- **Network programming**
  - Socket Security
  - Denial Of Service Attacks
  - User input & Other

Designing

Two common security mistakes

- **Applications are designed by teams of people who do not understand security**
- **Adding security to the application as an afterthought:**
  - It is expensive
  - Adding security might change the way you’ve implemented features
  - Adding security might change the application interface

Why these mistakes are made?

- Security is boring
- Security is often seen as functionality disabler
- Security is difficult to measure
- Security is usually not the primary skill or interest of the designers and developers creating the product

Security Principles to Live By

- **Establish a security process**
  - Management control and revision control of specifications, code, documentation and tests
- **Define the product security goals**
- **Consider security as a product feature**
- **Learn from mistakes**
- **Use least privilege**
- **Use defense in depth**
  - Imagine your application is the last application standing, and every defensive mechanism protecting you has been destroyed

Security Principles to Live By

- **Assume external systems are insecure**
- **Plan on failure**
  - What happens if the firewall is breached? Death, taxes and computer system failure are all inevitable to some degree. Plan for the event.
- **Fail to a secure mode**
  - If the attacker knows that he can make your code fail, he can bypass security mechanisms because your failure mode is insecure

Security Principles to Live By

- **Employ secure defaults**
  - The less often used features should be off by default to reduce potential security exposure
- **Remember that security features != secure features**
- **Never depend on security through obscurity**
  - Always assume that the attacker has access to all source code and all designs

Three Final Points

- If you find a security bug fix it and go looking for similar issues in other parts of code
- Make the fix as close as possible to the location of vulnerability
- Cure the problem, not the symptoms

Security design by Threat Modelling

- Brainstorm the known threats to the system
- Rank the threats by decreasing risk
- Choose how to respond to the threats
- Choose techniques to mitigate the threats
- Choose the appropriate technologies from the identified techniques

Security design by Threat Modelling

- **Brainstorm the known threats to the system**
  - Two or three hours for the initial brainstorm meeting with group up to 10 people
  - Have one person lead the meeting – the most security savvy of the team
  - At least one member from each development discipline: design, coding, testing, documentation
  - The design and code changes are made after the meeting
  - Have a overall system architecture on the whiteboard

Security design by Threat Modelling

- **The STRIDE Threat Model – categories:**
  - Spoofing identity – illegally accessing and then using another user’s authentication information.
  - Tampering with data – malicious modification of data like unauthorized changes made to persistent data or data as it flows between two computers over an open network, such as Internet

Security design by Threat Modelling

- **The STRIDE Threat Model – categories cont.:**
  - Repudiation – they are associated with users who deny performing an action without other parties having any way to prove otherwise.
    Nonrepudiation is the ability of the system to counter repudiation threats.
    Example:
    If a user purchases an item, he might have to sign for the item upon receipt. The vendor can then use the signed receipt as evidence that the user did receive the package.

Security design by Threat Modelling

- **The STRIDE Threat Model – categories cont.:**
  - Information disclosure – exposure of information to individuals who are not supposed to have access to it
    Example:
    A user's ability to read a file that she was not granted access to and an intruder's ability to read data in transit between two computers.
  - Denial of service
  - Elevation of privilege
    Example: Obtaining root account

Security design by Threat Modelling

- **Items to Note While Thread Modeling**
  - Title – descriptive and short i.e.: „Attacker accesses a user’s shopping cart”
  - Threat type(s) – a threat can fall under multiple STRIDE categories
  - Target – which part of application is prone to the attack
  - Chance – chance of the threat to occur from 1 (greatest) to 10 (least)

Security design by Threat Modelling

- **Items to Note While Thread Modeling – cont.:**
  - Criticality – extent and severity of the damage (some data are invaluable) from 1 (least damage) to 10 (greatest damage)
  - Attack techniques – How would an attacker manifest the threat?
  - Mitigation techniques (optional) – What would mitigate such threat? How difficult it is to mitigate?
  - Mitigation status – Has the threat been mitigated? Valid entries are: Yes, No, Somewhat and Needs Investigating.
  - Bug number

Security design by Threat Modelling

- **Rank the threats by Decreasing Risk**
  - Risk = Criticality / Chance
- **Choose how to respond to the Threats**
  - Do nothing
  - Inform the user of the threat
    - Many users don't what the right decision is
    - Users will learn to ignore warnings if they come up to often.
  - Remove the problem
    - There is always the next version!
  - Fix the problem

Security design by Threat Modelling

- **Choose techniques to mitigate the Threats (techniques != technologies)**
  - Spoofing identity
    - Authentication (i.e. X.509 certificates, IPSec, HTTP Basic Authentication, Digest Authentication, DCOM)
    - Protect secrets
    - Don’t store secrets
  - Tampering with data
    - Authorization (i.e. ACL, Privileges, IP adress restrictions)
    - Hashes
    - Message authentication codes
    - Digital signatures
    - Tamper-resistant protocols (SSL/TLS, IPSec, DCOM, EFS)

Security design by Threat Modelling

- **Choose techniques to mitigate the Threats – cont.:**
  - Repudiation
    - Digital signatures
    - Timestamps
    - Audit trails
  - Information disclosure
    - Authorization
    - Privacy-enhanced protocols
    - Encryption
    - Protect secrets
    - Don’t store secrets

Security design by Threat Modelling

- **Choose techniques to mitigate the Threats – cont.:**
  - Denial of service
    - Authentication
    - Authorization
    - Filtering
    - Throttling – limiting the number of requests to the system
    - Quality of service – i.e. prioritising the traffic
  - Elevation of privilege
    - Run with least privilege

Security design by Threat Modelling

- **Defining ACL (Access Control List) Lists:**
  - Determine the resource you use
  - Determine the business defined access requirements
  - Determine the appropriate access control technology
  - Convert the access requirements to access control technology

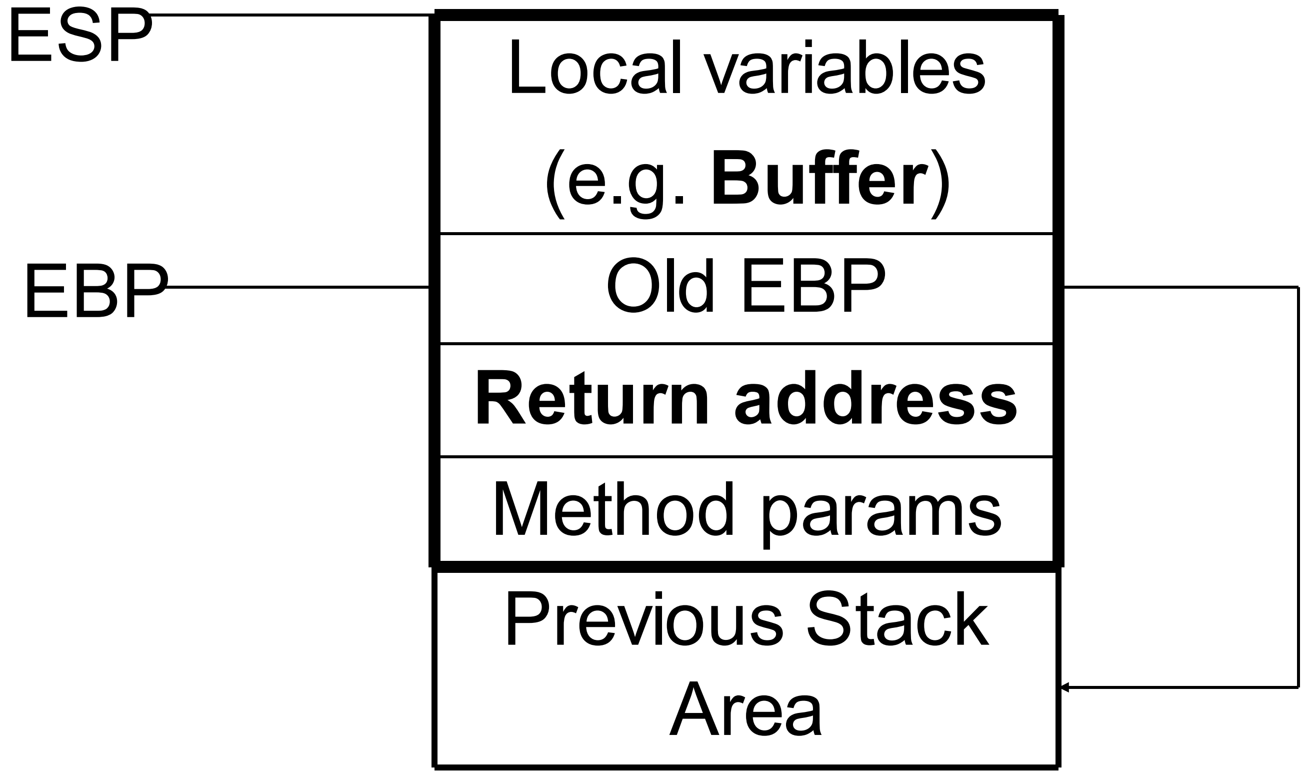
Security design by Threat Modelling

- **ACL consists of ACEs**
- **ACE is:**
  - Subject
  - Access Rights
- **Always place deny ACEs at the start of the ACL**

Programming

- **Public Enemy #1 – Buffer Overrun**
  - Static Buffer Overruns
  - Heap Overruns
  - Format String Bugs
  - Preventing Buffer Overruns

Stack



32-bit Intel processor

Static Buffer Overruns

- A buffer declared on the stack is overwritten by copying data larger than the buffer
- Variables declared on the stack are located next to the return address for the function's caller
- the return address for the function gets overwritten by an address chosen by the attacker

Static Buffer Overruns - example

- This program shows an example of how a static buffer overrun can be used to execute arbitrary code.
- Its objective is to find an input string that executes the function bar and execute the function bar

Static Buffer Overruns - example

A trick to view the stack

```
#include <stdio.h>
#include <string.h>

void foo(const char* input) {
  char buf[10];
  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
  strcpy(buf, input);
  printf("%s\n", buf);
  printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void) {
  printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[]) {
  printf("Address of foo = %p\n", foo);
  printf("Address of bar = %p\n", bar);
  foo(argv[1]);
  return 0;
}
```

Static Buffer Overruns - example

```
#include <stdio.h>
#include <string.h>

void foo(const char* input) {
  char buf[10];
  printf("My stack looks l                 n%p\n%p\n%p\n\n");
  strcpy(buf, input);
  printf("%s\n", buf);
  printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void) {
  printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[]) {
  printf("Address of foo = %p\n", foo);
  printf("Address of bar = %p\n", bar);
  foo(argv[1]);
  return 0;
}
```

Pass the user input without range checking.

Static Buffer Overruns - example

```
#include <stdio.h>
#include <string.h>

void foo(const char* input) {
  char buf[10];
  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
  strcpy(buf, input);
  printf("%s\n", buf);
  printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void) {
  printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[]) {
  printf("Address of foo = %p\n", foo);
  printf("Address of bar = %p\n", bar);
  foo(argv[1]);
  return 0;
}
```

To make life easier on myself

Static Buffer Overruns - example

```
StaticOverrun.exe Hello
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EDE
```

The return address of foo

```
Hello
Now the stack looks like:
6C6C6548
0000006F
7FFDF000
0012FF80
0040108A
00410EDE
```

„Hello” was copied in.

Static Buffer Overruns - example

```
StaticOverrun.exe ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EDE

Hello
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
54535251
58575655
```

The return address of foo

The NEW return address of foo

Static Buffer Overruns - example

- The application error message now shows that we’re trying to execute instructions at 0x54535251.
- Glancing at our ASCII charts, we see that 0x54 is the code for the letter T, so that’s what we’d like to modify ☺

Heap Overruns

- As in the case of a static buffer overrun, attacker can write fairly arbitrary information into places in application that he shouldn’t have access to
- Many programmers don’t think heap overruns are exploitable, leading them to handle allocated buffers with less care than static buffers

Heap Overruns - example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class BadStringBuf {
public:
    BadStringBuf(void) { m_buf = NULL; }
    ~BadStringBuf(void) {
        if(m_buf != NULL)
            free(m_buf);
    }
    void Init(char* buf) { m_buf = buf; }
    void SetString(const char* input) { strcpy(m_buf, input); }
    const char* GetString(void) { return m_buf; }

private:
    char* m_buf;
};
```

Heap Overruns – example cont.

```
BadStringBuf* g_pInput = NULL;

void bar(void) { printf("Augh! I've been hacked!\n"); }

void BadFunc(const char* input1, const char* input2) {
    char* buf = NULL;
    char* buf2 = (char*) malloc(16);

    g_pInput = new BadStringBuf;
    buf = (char*) malloc(16);
    g_pInput->Init(buf2);

    strcpy(buf, input1);
    g_pInput->SetString(input2);

    printf("input 1 = %s\ninput2 = %s\n", buf, g_pInput->GetString());

    if(buf != NULL)
        free(buf);
}
```

No error checking on allocations

Heap Overruns – example cont.

- Application blows up whether the first or second argument becomes to long but that the address of the error shows that the memory corruption occurs on the heap
- Using the debugger you find out that the pointer to the second buffer is sitting just a 0x40 bytes past the location from the 1st buffer start – so we can change the pointer address of the buffer to point on stack

Heap Overruns – example cont.

```
int main(int argc, char* argv[]) {
    char arg1[128];
    char arg2[4] = {0x0f, 0x10, 0x40, 0};
    int offset = 0x40;

    memset(arg1, 0xfd, offset);
    arg1[offset]   = (char) 0x94;
    arg1[offset+1] = (char) 0xfe;
    arg1[offset+2] = (char) 0x12;
    arg1[offset+3] = 0;
    arg1[offset+4] = 0;

    printf("Address of bar is %p\n", bar);
    BadFunc(arg1, arg2);

    if(g_pInput != NULL)
        delete g_pInput;

    return 0;
}
```

The address of the bar function.

To overcome heap corruption checking

The return address in the stack

Integer Overflow

- Overflow and underflow
- Signed versus unsigned
- Truncation

Integer Overflow

- Overflow and underflow

```
bool func(size_t cbSize) {
    if (cbSize < 1024) {
        char *buf = new char[cbSize-1];
        if (buf == NULL) return false;
        memset(buf,0,cbSize-1);
        ...
        delete [] buf;
        return true;
    } else {
        return false;
    }
}
```

Integer Overflow

- Signed versus unsigned

```
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;
}
```

Integer Overflow

- Truncation

```
bool func(byte *name, DWORD bufSize) {
    unsigned short calculatedBufSize = bufSize;
    byte *buf = (byte*) malloc(calculatedBufSize);

    if (buf) {
        memcpy(buf, name, bufSize);
        ...
        if (buf) free(buf);
        return true;
    }
    return false;
}
```

Format string bugs

The coding construct in DCOM that led to the Blaster worm

```
HRESULT GetMachineName(WCHAR *pwszPath,
    WCHAR wszMachineName[N + 1]) {

    LPWSTR pwszServerName = wszMachineName;
    while (*pwszPath != '\\')
        *pwszServerName++ = *pwszPath++;
    ...
}
```

Format string bugs

Secured

```
HRESULT GetMachineName(WCHAR *pwszPath,
    WCHAR wszMachineName[N + 1]) {

    LPWSTR pwszServerName = wszMachineName;

    size_t machineName = N;
    while (*pwszPath != '\\' && --machineName)
        *pwszServerName++ = *pwszPath++;
    ...
}
```

Preventing buffer overruns

- Always validate all your inputs and outputs
- Safe String Handling

```
bool HandleInput(const char* input) {
    char buf[80];

    if(input == NULL) {
        assert(false);
        return false;
    }
    if(strlen(input) < sizeof(buf)) {
        strcpy(buf, input);
    } else {
        return false;
    }

    return true;
}
```

Preventing buffer overruns

- **Safe String Handling – cont.**

```
bool HandleInput_Strncpy2(const char* input) {
    char buf[80];
    if(input == NULL) {
        assert(false);
        return false;
    }
    buf[sizeof(buf) - 1] = '\0';
    strncpy(buf, input, sizeof(buf));
    if(buf[sizeof(buf) - 1] != '\0') {
        //Overflow!
        return false;
    }
    return true;
}
```

Preventing buffer overruns

Spot the security flaw

```
bool SprintfLogError(int line, unsigned long err,
    char* msg) {

    char buf[132];
    if(msg == NULL) {
        assert(false);
        return false;
    }

    sprintf(buf, "Error in line %d = %d - %s\n", line,
        err, msg);

    return true;
}
```

Preventing buffer overruns

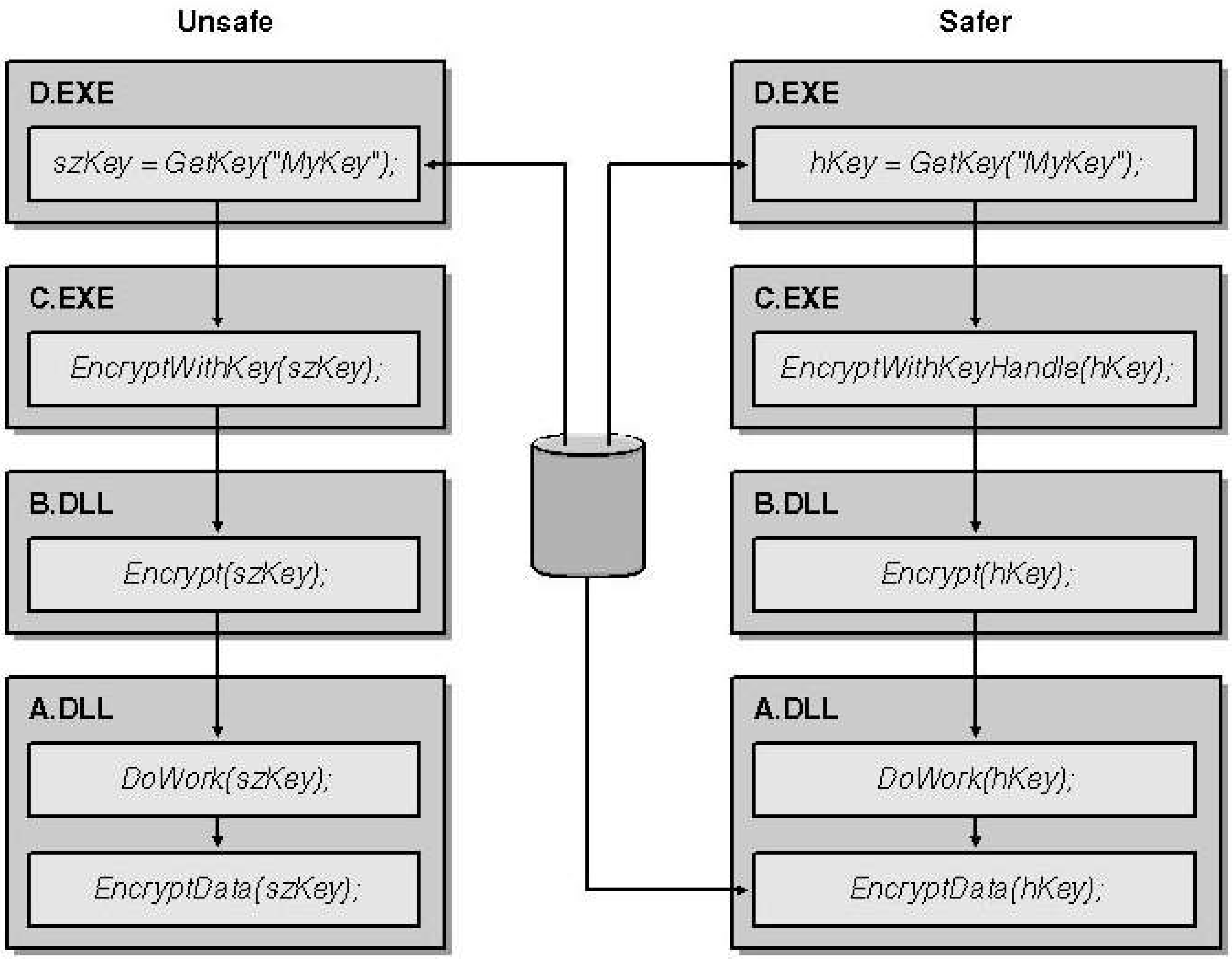
- **Standard Template Library Strings**

```
#define MAX_CHARS  132
#include <string>
using namespace std;

void HandleInput_STL(const char* input) {
    string str1, str2;

    str2.append(input, MAX_CHARS);

    printf("%s\n", str2.c_str());
}
```

Cryptographic Foibles

- Don’t use int rand(void) to create passwords etc. because of its predictability
- Don’t use your own cryptographic functions
- Keep keys close to the source

Cryptographic Foibles – cont.



Storing secrets

- **Threats**
  - Attacker can debug process using the secret, set a breakpoint at the location where the code gathers the information and read the data in the debugger
  - The memory holding the secret becomes paged to the page file

Storing secrets – cont.

- **Prevention**
  - Store the (salted) hash of the secret, not the secret itself
  - Get the secret from the user each time the secret is used
  - Use external devices to encrypt secret data

Canonical Representation Issues

- There is often more than one valid way to represent the resource name
- Example

```
C:\temp\longfilename.txt
C:\temp\longfi~1.txt
\\?\C:\temp\longfilename.txt
C:\temp\..\temp\longfilename.txt
```

Canonical Representation Issues

- **Web-specific bugs for web pages and urls**
  - The “normal” 7-bit or 8-bit character representation
  - Hexadecimal escape codes
  - UTF-8 variable-width encoding
  - UCS-2 Unicode encoding
  - Double encoding
  - HTML escape codes (Web pages, not URLs)
  - „Parent Paths”
    - Example: <img src=„../images/logo.jpg”>
      Attacker can access data outside of the Web root

Canonical Representation Issues

- **Preventing**
  - Don’t make decisions based on names
  - Use regular expression to restrict what’s allowed in a name
  - Stopping 8.3 file name generation
  - Don’t trust the PATH variable
  - Canonicalize the name
  - Design web-based system in such a way that parent paths are not required

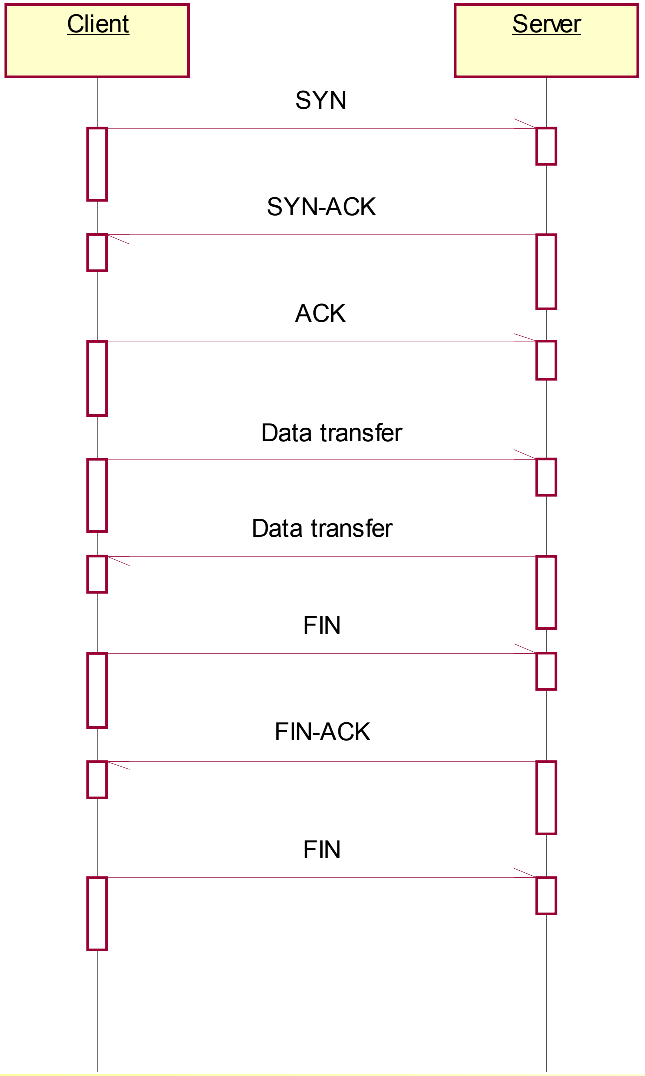
Network programming

- **Socket Security**
  - Avoid server hijacking by binding the address with SO_EXCLUSIVEADDRUSE
  - Any IP service should be configurable at one of three levels:
    - Which network interface is listening
    - Which IP address(es) it listens on and on which port
    - Which clients can connect to the service

Network programming

- **Socket Security**
  - Accepting connections
    - Connectionless (UDP)
      - Drop the packet and don’t send a reply if you don’t want to accept the request
      - A reply consumes your resources and gives the attacker information

Connection-based (TCP)



Connection-based (TCP)

- Even if You immediately drop the connection, the attacker knows that some service is listening on that port
- We’re also going to exchange a total of seven packets in the process of telling the client to go away
- he might have hacked his IP stack to never send the FIN-ACK in response to our FIN. If that’s the case, we’ll wait two segment lifetimes for a reply

Connection-based (TCP)

- **There are system solutions for conditional acceptance of the connection (SO_CONDITIONAL_ ACCEPT)**
  - You decide whether or not to establish connection with the client
  - It sends 3 times SYN packet to the client

Firewall-Friendly Applications

- Use one connection to do the job
- Don’t make connections back to the client from the server
- Connection-based protocols are easier to secure
- Don’t try to multiplex your application over another protocol
- Don’t embed host IP addresses in application-layer data
- Configure your client and server to customize the port used

Spoofing

- **Three hosts:**
  - An attacker
  - A victim
  - Innocent third party
- **Don’t rely on ip address or dns name – prove it with the shared secret, a certificate or other strong crypt method**

User input

- **Mistake #1 Trusting the user**
  - A Web service that allows users a guestbook
  - The attacker doesn’t like the guestbook
  - The attacker posts the message

  <meta http-equiv=”refresh” content=”2;URL=http://www.google.pl/”>
  - Every time someone will look through the guestbook he will be presented the www.google.pl page
- **Mistake #2 Unbounded Sizes – buffer underrun**

User input

- **Mistake #3 Using Direct User Input in SQL Statements**
  - The attacker knows a username and wants to spoof that user account
  - The SQL statement checking user name and password looks like this:

```
SELECT count(*) FROM client
WHERE name=$NAME and pwd=$PWD
```

User input

- **Mistake #3 Using Direct User Input in SQL Statements**

  The attacker enters name `Cheryl' –`

  The SQL statement will look:

```
SELECT count(*)
FROM client
WHERE name='Cheryl' --and pwd=''
```

  The SQL statements can be joined

```
SELECT * from client INSERT into client
  VALUES ('me', 'URHacked')
```

User input

- **Mistake #4 Not being strict is dangerous**
  - Many Web developers allow “safe” HTML constructs
  - The user can send HTML tags but nothing else, other than plaintext. A cross-site scripting danger still exists because the attacker can embed script in some of these tags. Here are some examples:
    - <img src=javascript:alert(document.domain)>
    - <link rel=stylesheet href="javascript:alert (document.domain)">
    - <input type=image src=javascript:alert (document.domain)>

User input & Others

- **Never trust user input**
  - All input is bad until proven otherwise
  - Data must be validated as it crosses the boundary between untrusted and trusted environments
  - All the dangerous characters must be converted to their safe equivalent or removed from the input
- **Do not use password directly**
- **Don’t store secrets in web pages**
- **Use HTTP GET only for queries**

Literature

- Malicious html tags
  - www.cert.org/advisories/CA-2000-02.html
- Phrack
  - www.phrack.org
- SQL Injection Wargame
  - http://www.hackingzone.org/sql/index.php
- Try 2 Hack
  - http://www.try2hack.nl/
- M.Howard, D. LeBlanc: Bezpieczny kod – Tworzenie i zastosowanie, Microsoft Press 2002
- Simson Garfinkel i Gene Spafford, WWW Bezpieczeństwo i handel, Helion 1999

The end

- Thank You for Your attention ☺