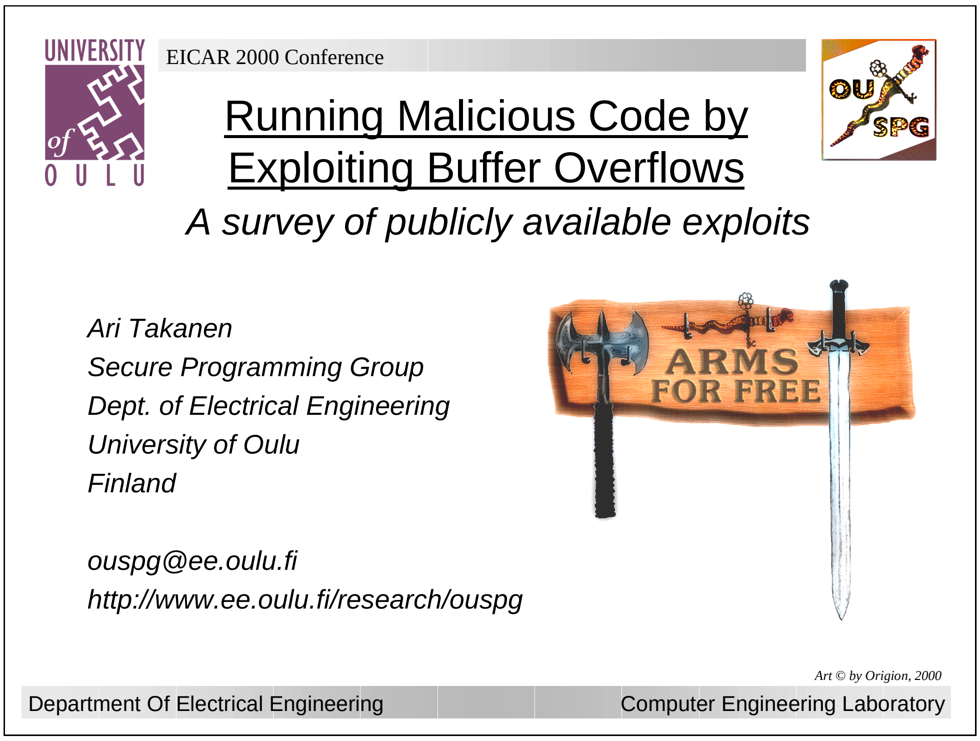


EICAR 2000 Conference

Running Malicious Code by Exploiting Buffer Overflows

A survey of publicly available exploits

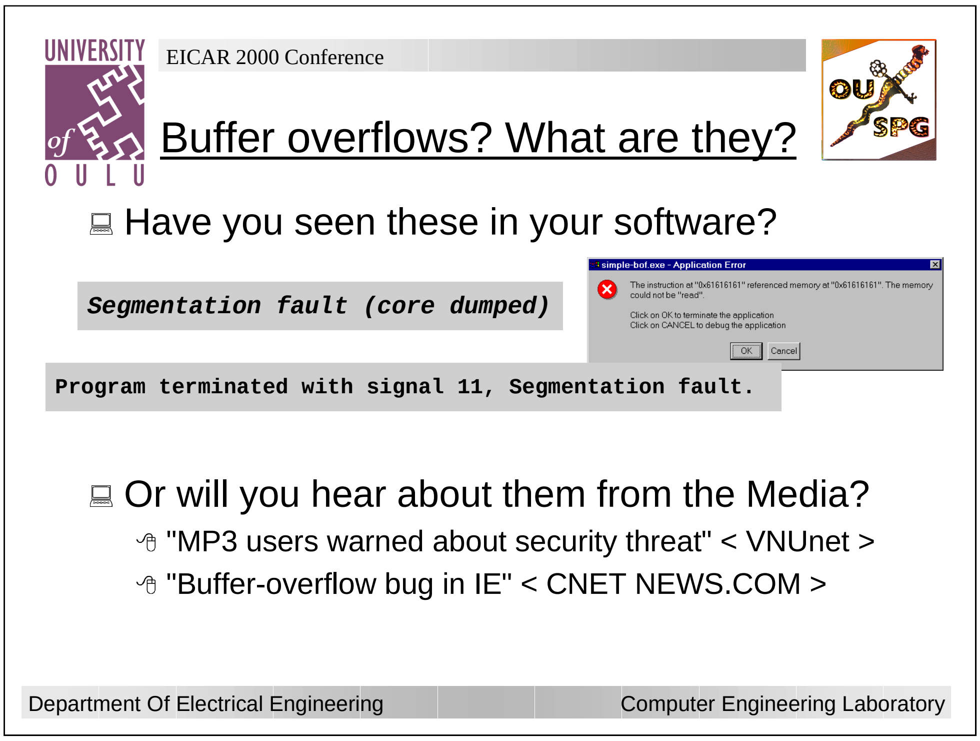
Ari Takanen
Secure Programming Group
Dept. of Electrical Engineering
University of Oulu
Finland

ouspg@ee.oulu.fi
http://www.ee.oulu.fi/research/ouspg

Art © by Origion, 2000

This presentation is about Buffer Overflows, and presents a case study that explores an archive of publicly available exploits against this type of vulnerability.

I hope you find the accompanying paper useful in beginning or continuing your research in this field.



Buffer overflows? What are they?

- Have you seen these in your software?

```
Segmentation fault (core dumped)
```

simple-bof.exe - Application Error

The instruction at "0x61616161" referenced memory at "0x61616161". The memory could not be "read".

Click on OK to terminate the application
Click on CANCEL to debug the application

```
Program terminated with signal 11, Segmentation fault.
```

- Or will you hear about them from the Media?
  - "MP3 users warned about security threat" < VNUnet >
  - "Buffer-overflow bug in IE" < CNET NEWS.COM >

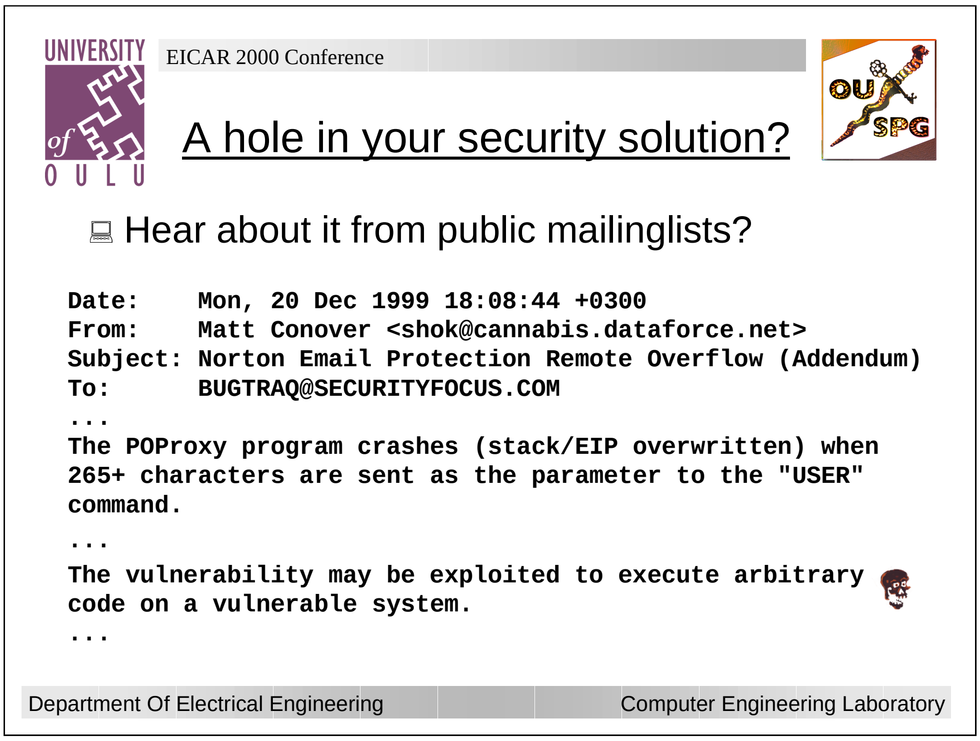
What are these buffer overflows anyway?

Usually buffer overflows are noticed when a software crashes with some note of violent memory accesses.

Developers might think the worst thing that can happen is the famous Denial of Service attack.

A bit much more scary way of getting introduced to buffer overflows is when it is too late.

Besides causing grief, the security vulnerabilities of this level of severity often gain the attention of the media.



A hole in your security solution?

- Hear about it from public mailinglists?

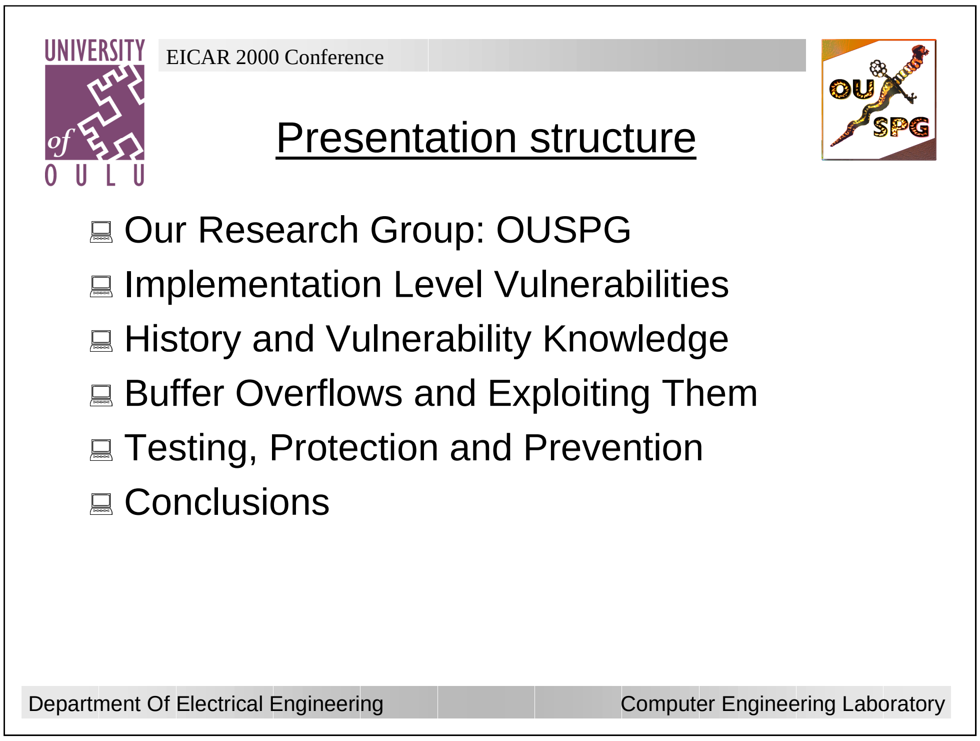
```
Date:    Mon, 20 Dec 1999 18:08:44 +0300
From:    Matt Conover <shok@cannabis.dataforce.net>
Subject: Norton Email Protection Remote Overflow (Addendum)
To:      BUGTRAQ@SECURITYFOCUS.COM
...
The POProxy program crashes (stack/EIP overwritten) when
265+ characters are sent as the parameter to the "USER"
command.
...
The vulnerability may be exploited to execute arbitrary
code on a vulnerable system.
...
```

Some emaillists such as Bugtraq are also fond of buffer overflow vulnerabilities.

Here is one discovered overflow from a popular security solution.

Quality should be required at least from protective software!

The key words about buffer overflows in this any many other public notices is that the overflow can be "… exploited to execute arbitrary code on a vulnerable program."



Presentation structure

- Our Research Group: OUSPG
- Implementation Level Vulnerabilities
- History and Vulnerability Knowledge
- Buffer Overflows and Exploiting Them
- Testing, Protection and Prevention
- Conclusions

This talk has the following structure:

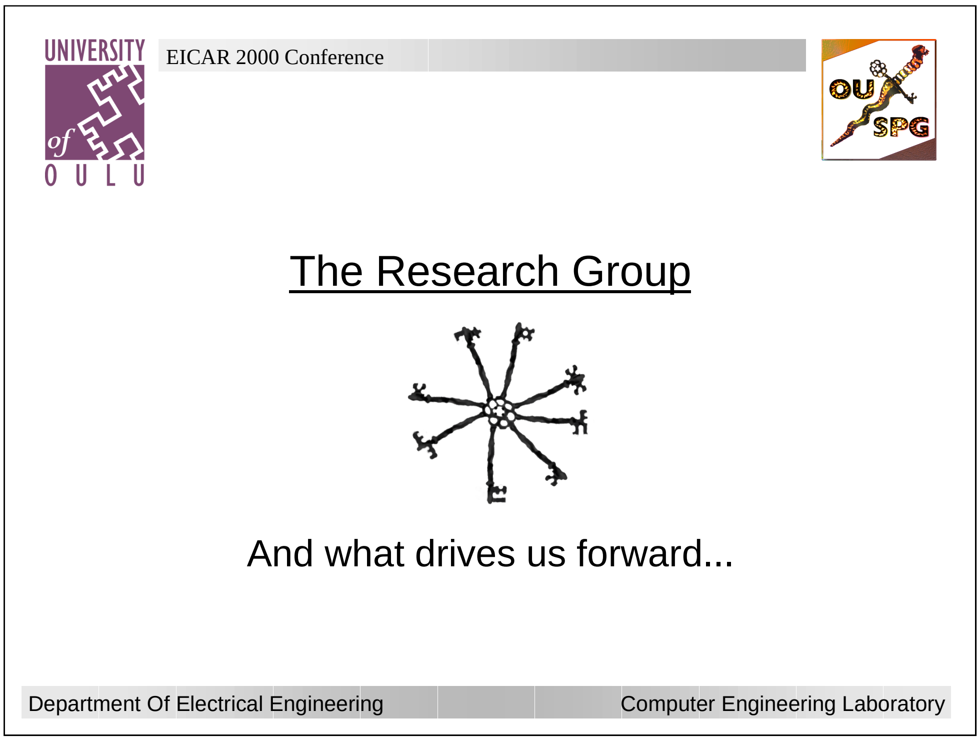
I will start by giving a short presentation of our research group and what drives us forward, followed by our major topic of research, the Implementation Level Vulnerabilities.

Then I will give some background information on some sources that I have used as references, and I will introduce the related case study.

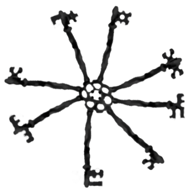
I will continue to buffer overflow details, and the operation of the stack, drawing examples from the exploits of the case study.

Afterwards I will present some methods of improving the security by testing or system modifications.

The concluding slides will summarize some major points from the presentation and the paper.

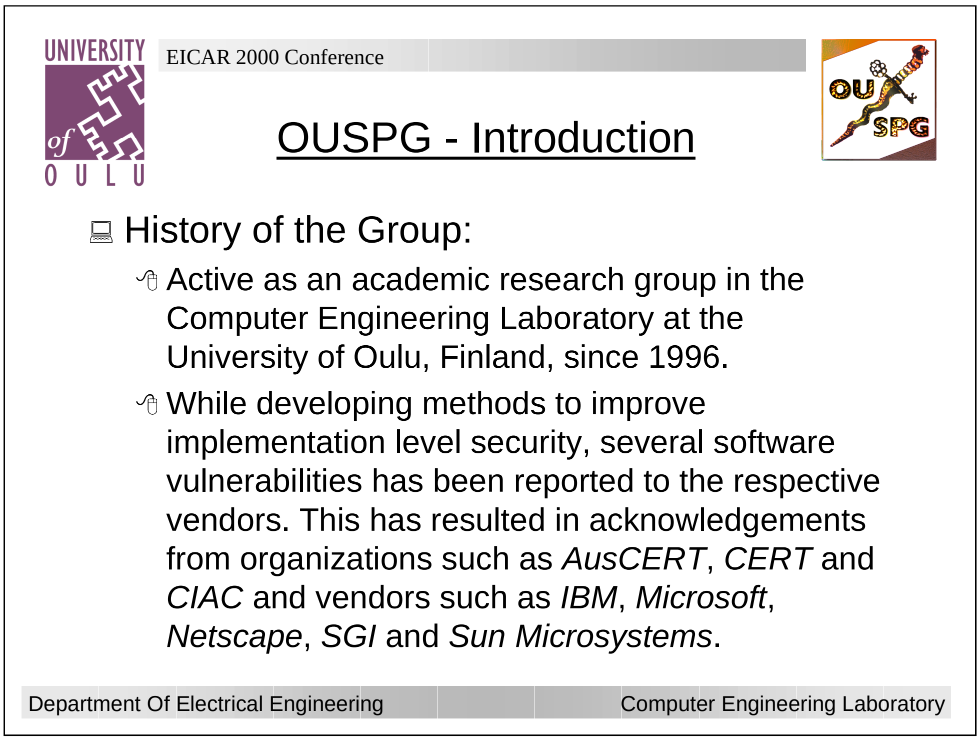


The Research Group



And what drives us forward...

I will begin by shortly introducing our research group.

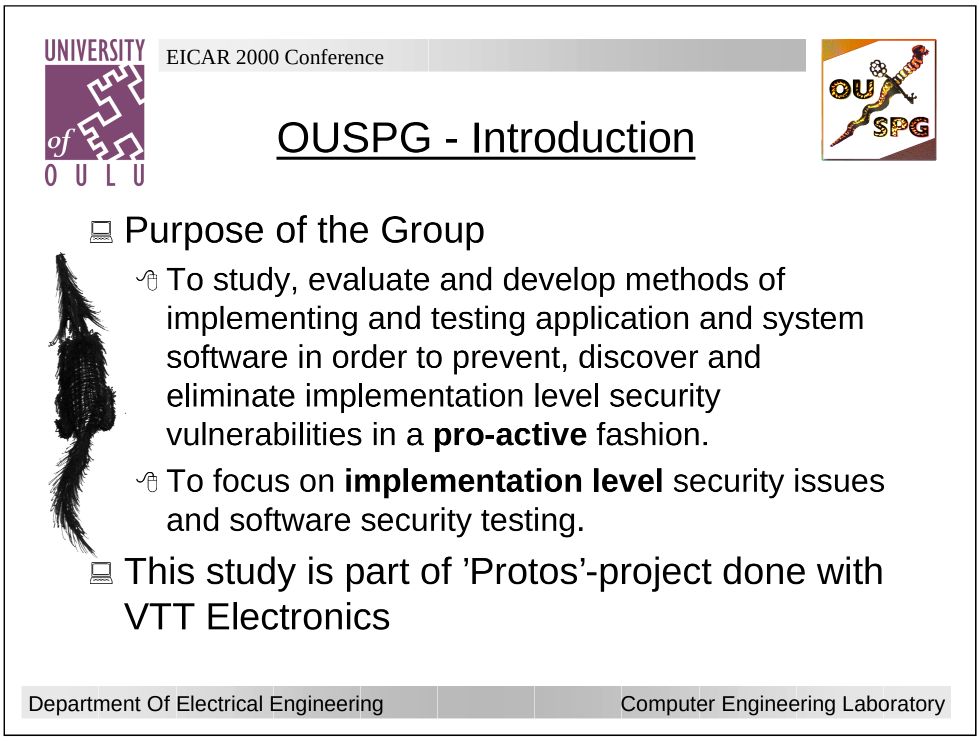


OUSPG - Introduction

- History of the Group:
  - Active as an academic research group in the Computer Engineering Laboratory at the University of Oulu, Finland, since 1996.
  - While developing methods to improve implementation level security, several software vulnerabilities has been reported to the respective vendors. This has resulted in acknowledgements from organizations such as *AusCERT*, *CERT* and *CIAC* and vendors such as *IBM*, *Microsoft*, *Netscape*, *SGI* and *Sun Microsystems*.

The research group has been active in the security field since 1996, beginning its existence from finding some security bugs 'by mistake' when administering the computers of the department.

Soon afterwards, while researching for possible methods of finding these vulnerabilities, a steady flow of vulnerability reports has been sent out from our group to relevant developers and coordinating entities.


EICAR 2000 Conference

OUSPG - Introduction

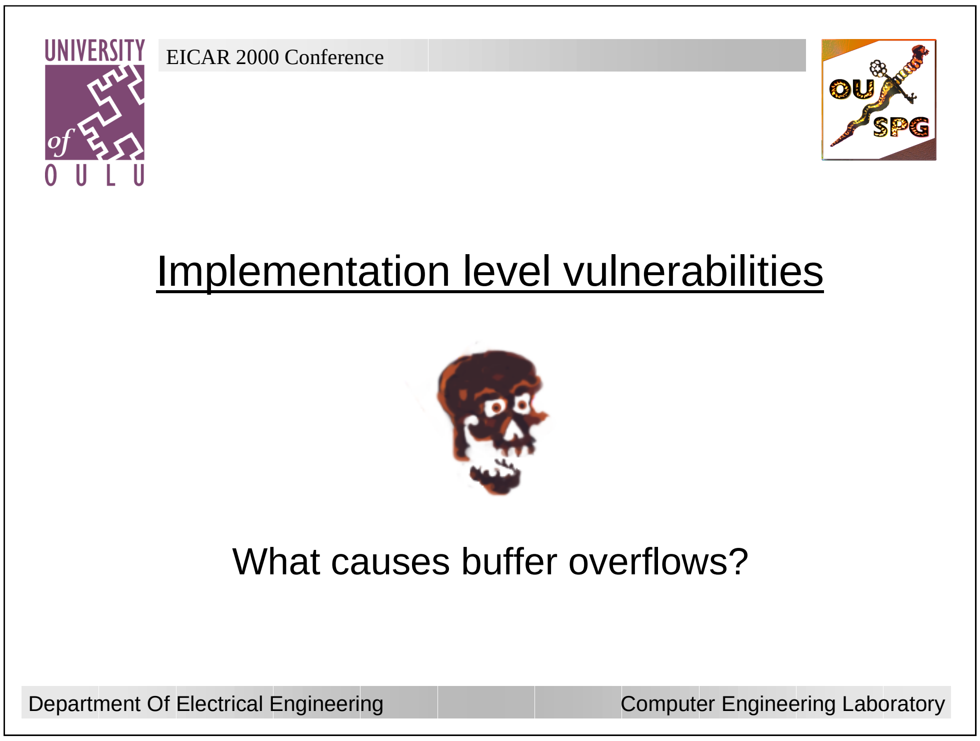
- Purpose of the Group
  - To study, evaluate and develop methods of implementing and testing application and system software in order to prevent, discover and eliminate implementation level security vulnerabilities in a **pro-active** fashion.
  - To focus on **implementation level** security issues and software security testing.
- This study is part of ’Protos’-project done with VTT Electronics

Department Of Electrical Engineering Computer Engineering Laboratory

This brings us to the major focus of the group, which is the development of proactive methods of searching for implementation level security issues.

The methods developed consist of testing tools, both for instrumenting the targets of testing, and framework for generating the testcases.

The development of the testing tools for the framework are currently done in close cooperation with the VTT Electronics, which is a part of the Technical Research Centre of Finland. This paper is part of this project.



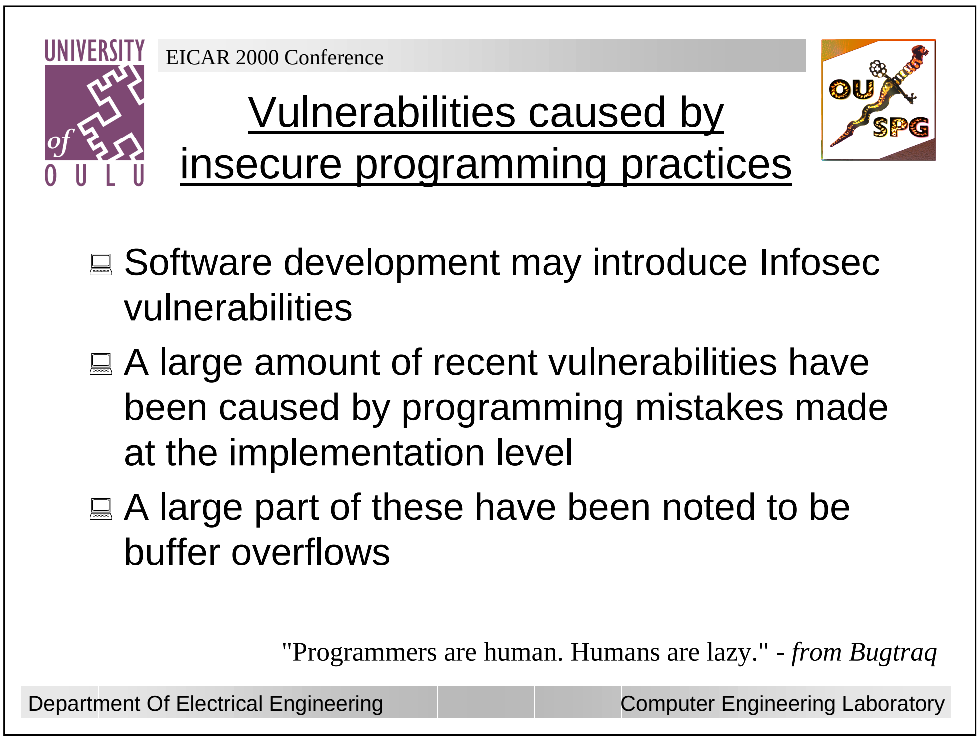
EICAR 2000 Conference

Implementation level vulnerabilities

What causes buffer overflows?

Department Of Electrical Engineering Computer Engineering Laboratory

Next I will show you some reasons behind these vulnerabilities that we are interested in.



UNIVERSITY of OULU

EICAR 2000 Conference

Vulnerabilities caused by insecure programming practices

OU SPG

- Software development may introduce Infosec vulnerabilities
- A large amount of recent vulnerabilities have been caused by programming mistakes made at the implementation level
- A large part of these have been noted to be buffer overflows

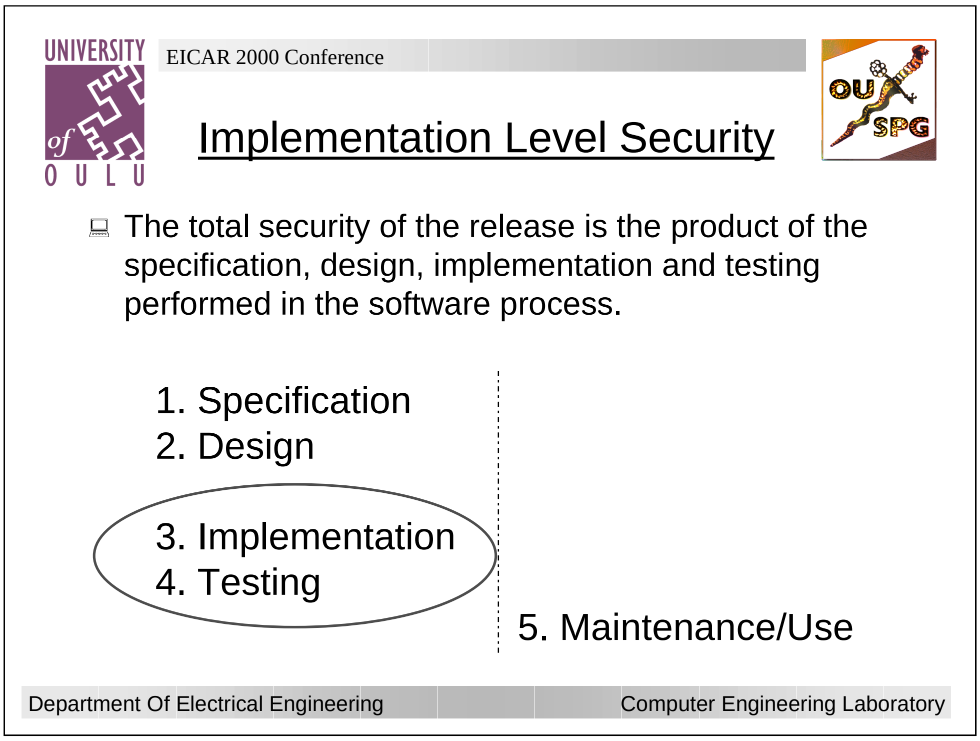
"Programmers are human. Humans are lazy." - *from Bugtraq*

Department Of Electrical Engineering Computer Engineering Laboratory

Software development may introduce information security vulnerabilities.

Often, these software vulnerabilities are caused by a programming mistakes.

Buffer overflow is one of these Implementation level vulnerabilities.



Implementation Level Security

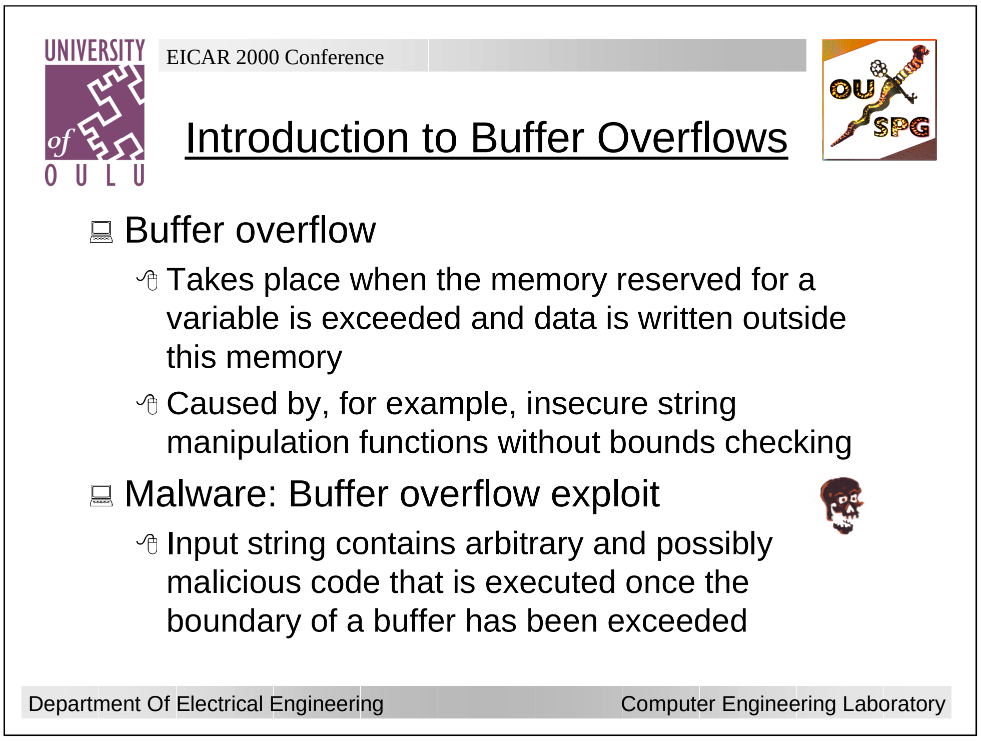
- The total security of the release is the product of the specification, design, implementation and testing performed in the software process.

1. Specification
2. Design
3. Implementation
4. Testing
5. Maintenance/Use

The focus of our research concentrates on these information security vulnerabilities caused by programming faults, that should have been caught at the testing phase.

We will try to avoid the wide field of security development done in the specification and design phase.

Another large category where vulnerabilities are introduced is the maintenance and configuration phase. These two are way out of scope for our research.



EICAR 2000 Conference

Introduction to Buffer Overflows

- Buffer overflow
  - Takes place when the memory reserved for a variable is exceeded and data is written outside this memory
  - Caused by, for example, insecure string manipulation functions without bounds checking
- Malware: Buffer overflow exploit
  - Input string contains arbitrary and possibly malicious code that is executed once the boundary of a buffer has been exceeded

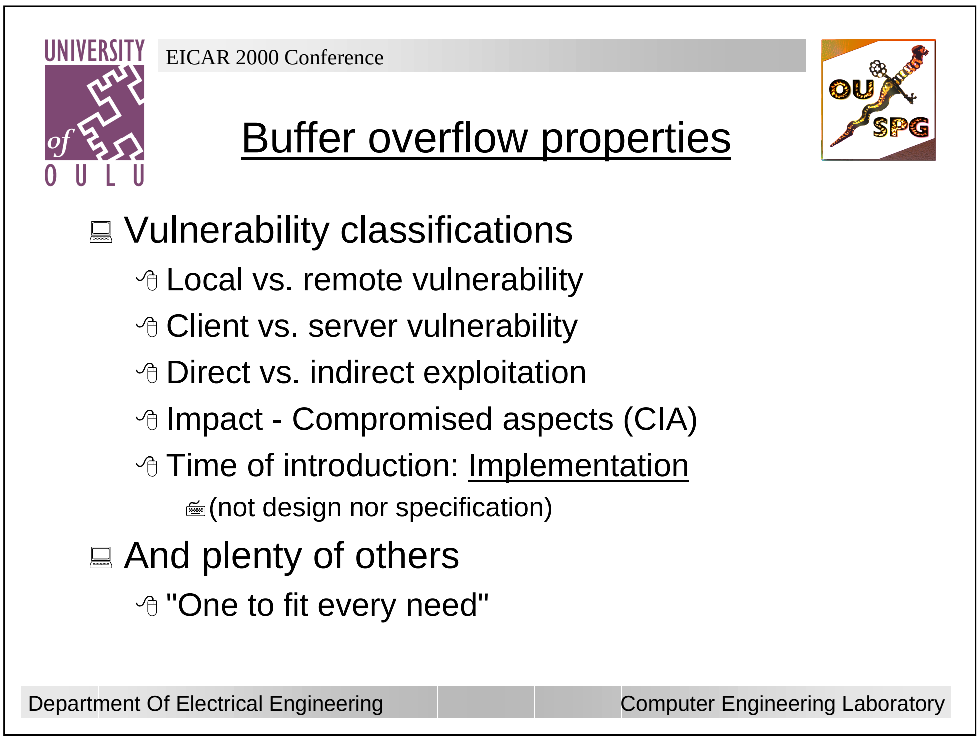
In short, buffer overflow happens when the input is stored without checking its size, in a place where too small a memory is reserved for storing it.

This is usually caused by insecure programming practices or wrong choice of function for a specific purpose.

Programming languages without build-in bounds-checking, like C and C++, contain several functions that introduce buffer overflow vulnerabilities.

A piece of code or a script that exploits the vulnerability is typically just called an exploit.

In a buffer overflow exploit, the input string contains data that takes the control from the target software to perform the wanted operations. Arbitrary and possibly malicious code can be executed with the privileges of the victim.



EICAR 2000 Conference

Buffer overflow properties

- Vulnerability classifications
  - Local vs. remote vulnerability
  - Client vs. server vulnerability
  - Direct vs. indirect exploitation
  - Impact - Compromised aspects (CIA)
  - Time of introduction: <u>Implementation</u>
    - (not design nor specification)
- And plenty of others
  - "One to fit every need"

Department Of Electrical Engineering Computer Engineering Laboratory

There exists several different ways of trying to categorize information security vulnerabilities.

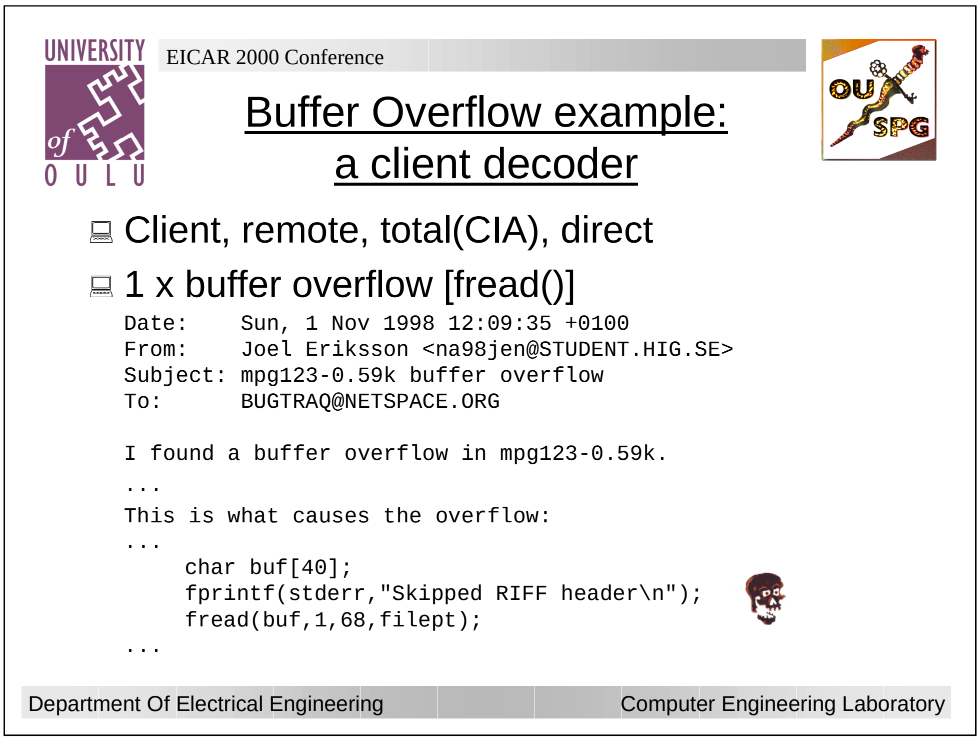
Buffer overflows do not seem to fit into any other categorization besides the time of introduction of the vulnerability.

As we know, there are both remote and local exploits for overflows.

Also, both clients and servers have been noted to be vulnerable for buffer overflows.

Some buffer overflows require indirect introduction of the exploit.

Anyway, all of the buffer overflows are introduced by programming ’faults’.



Buffer Overflow example: a client decoder

- Client, remote, total(CIA), direct
- 1 x buffer overflow [fread()]

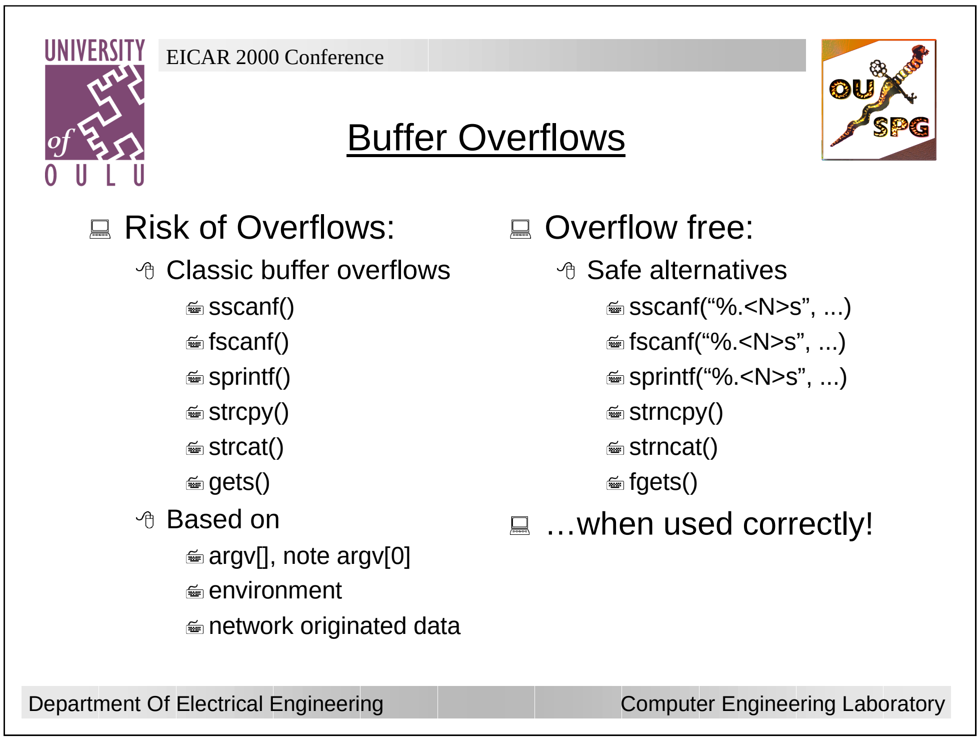
```
Date:    Sun, 1 Nov 1998 12:09:35 +0100
From:    Joel Eriksson <na98jen@STUDENT.HIG.SE>
Subject: mpg123-0.59k buffer overflow
To:      BUGTRAQ@NETSPACE.ORG

I found a buffer overflow in mpg123-0.59k.
...
This is what causes the overflow:
...
     char buf[40];
     fprintf(stderr,"Skipped RIFF header\n");
     fread(buf,1,68,filept);
...
```

Here is a simple example of a public announcement of a buffer overflow, with example of the vulnerable code.

This example shows an overflow in a client software for decoding audio-stream. It can be exploited remotely, and provides total control on the victim, violating all Confidentiality, Integrity and Availability.

This, like most buffer overflows, is directly exploitable, without any other additional steps.


EICAR 2000 Conference

Buffer Overflows

- Risk of Overflows:
  - Classic buffer overflows
    - sscanf()
    - fscanf()
    - sprintf()
    - strcpy()
    - strcat()
    - gets()
  - Based on
    - argv[], note argv[0]
    - environment
    - network originated data
- Overflow free:
  - Safe alternatives
    - sscanf(“%.<N>s”, ...)
    - fscanf(“%.<N>s”, ...)
    - sprintf(“%.<N>s”, ...)
    - strncpy()
    - strncat()
    - fgets()
- …when used correctly!

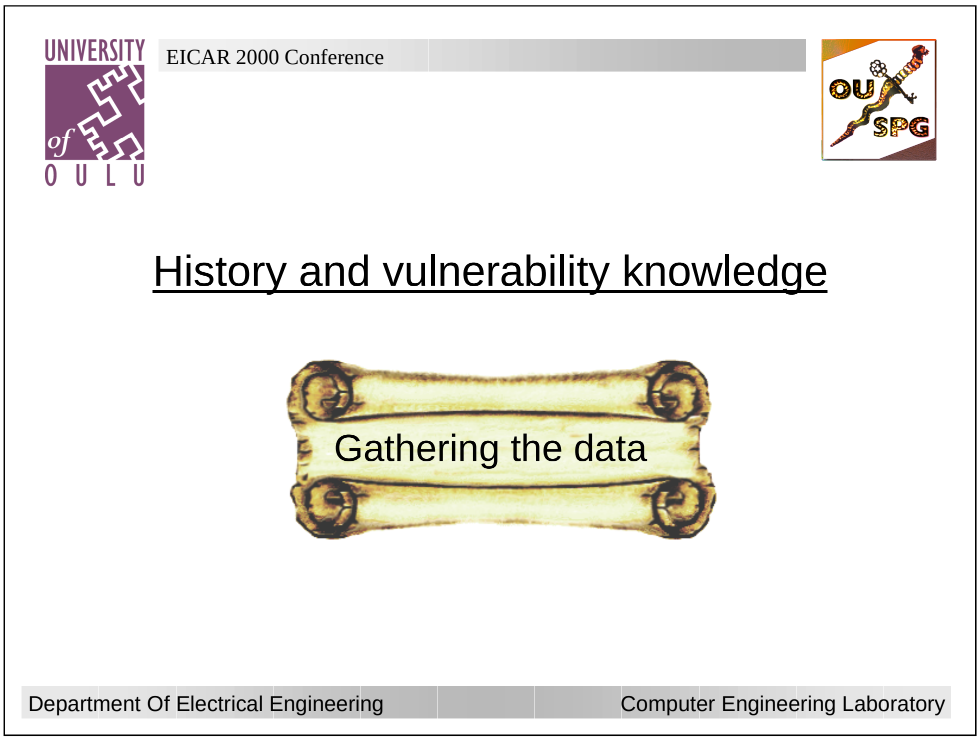
Department Of Electrical Engineering Computer Engineering Laboratory

This slide shows a summary of the best known functions behind these buffer overflow vulnerabilities.

As this slide shows, there usually is a secure option for the insecure functions.

Every programmer making security critical code should be aware of the risks of using the insecure functions, and should be told to use the secure one instead.

Still, the secure choices can be abused as well, as the programmers often find ways of creating insecure code even with the more secure functions.



EICAR 2000 Conference

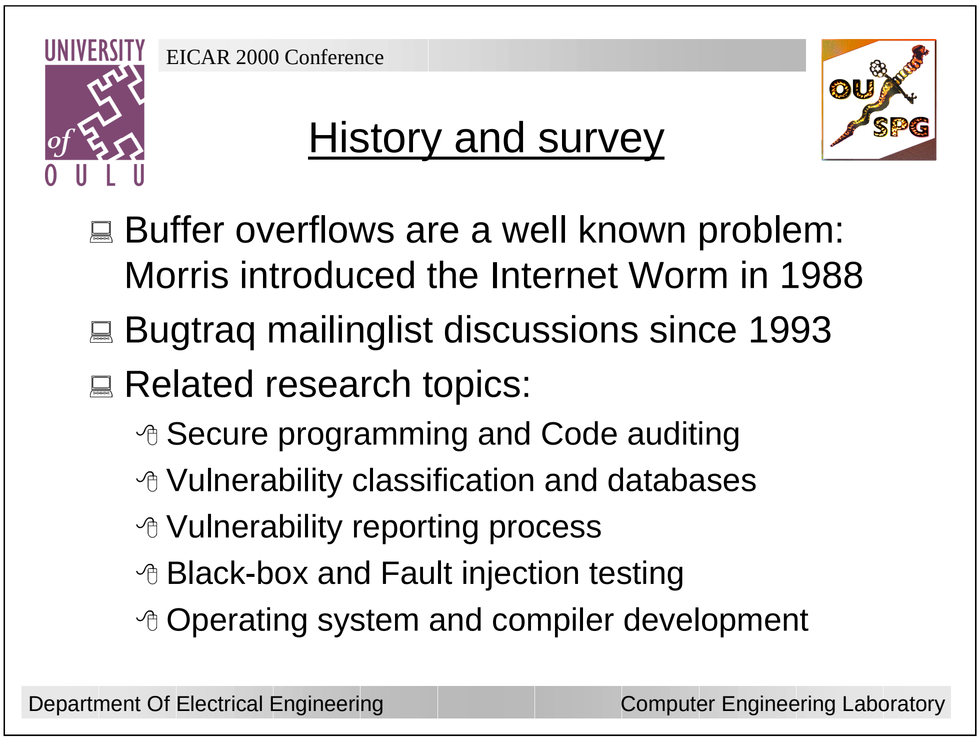
History and vulnerability knowledge

Gathering the data

Department Of Electrical Engineering

Computer Engineering Laboratory

Next we will have a look into the history of buffer overflows, and quickly view the different sources of information on overflow vulnerabilities.



EICAR 2000 Conference

History and survey

- Buffer overflows are a well known problem: Morris introduced the Internet Worm in 1988
- Bugtraq mailinglist discussions since 1993
- Related research topics:
  - Secure programming and Code auditing
  - Vulnerability classification and databases
  - Vulnerability reporting process
  - Black-box and Fault injection testing
  - Operating system and compiler development

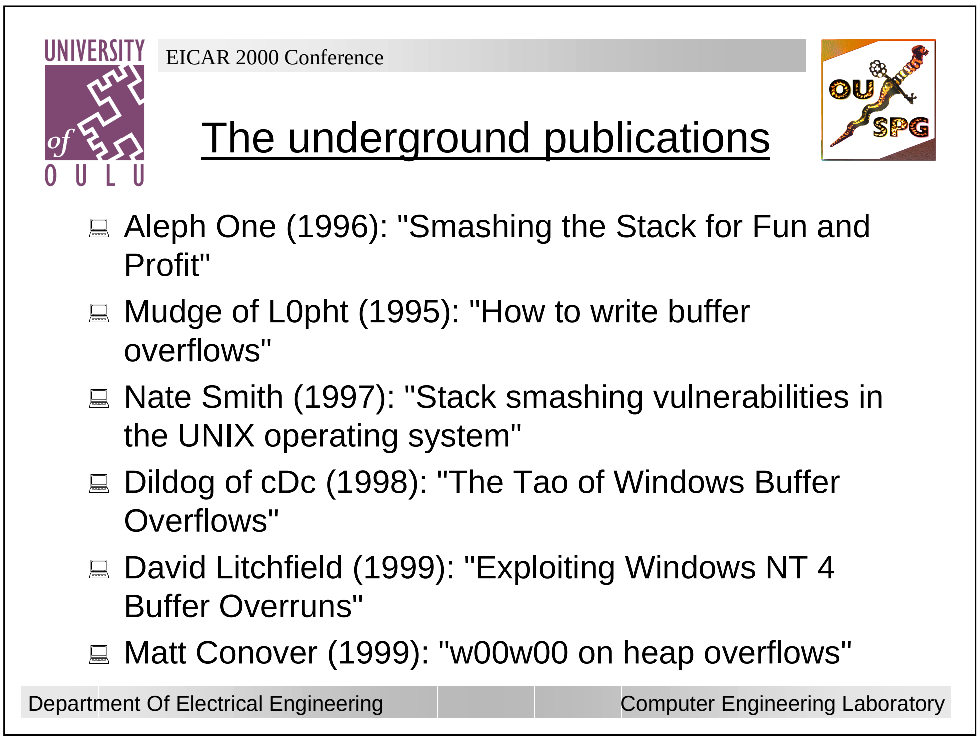
Department Of Electrical Engineering Computer Engineering Laboratory

As all implementation level vulnerabilities, also buffer overflows have been known for a long time.

In 1988, the world was shocked by an Internet Worm that used the vulnerability in finger daemon to spread around the Internet of that time, bringing it down to its knees.

With the introduction of open discussion channels like Bugtraq, the details were finally publicly available for the security community.

Also work on other related fields have been introduced, and are discussed in more detail in our paper. These include secure programming and code auditing, several classification studies, and studies of the whole vulnerability reporting process. Several methods of testing for vulnerabilities have been discussed as well as several improvements to operating systems to reduce the impact of these faults.



EICAR 2000 Conference

The underground publications

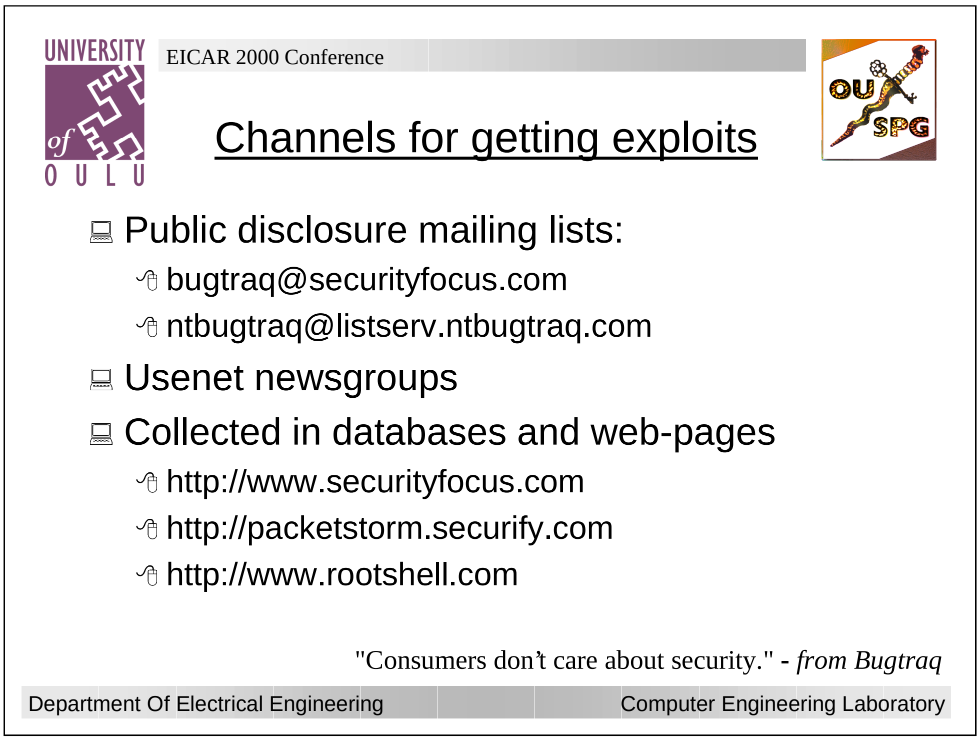
- Aleph One (1996): "Smashing the Stack for Fun and Profit"
- Mudge of L0pht (1995): "How to write buffer overflows"
- Nate Smith (1997): "Stack smashing vulnerabilities in the UNIX operating system"
- Dildog of cDc (1998): "The Tao of Windows Buffer Overflows"
- David Litchfield (1999): "Exploiting Windows NT 4 Buffer Overruns"
- Matt Conover (1999): "w00w00 on heap overflows"

Department Of Electrical Engineering Computer Engineering Laboratory

Perhaps a bit less academic approach has been taken by the security community, and several articles have appeared that discuss the details of exploiting the buffer overflow.

Since 1995, the articles have been public, and have drawn discussion on the subject. The paper gives a detailed study of these articles and the major points brought up in them.

I really suggest studying these further if this issue appears interesting.



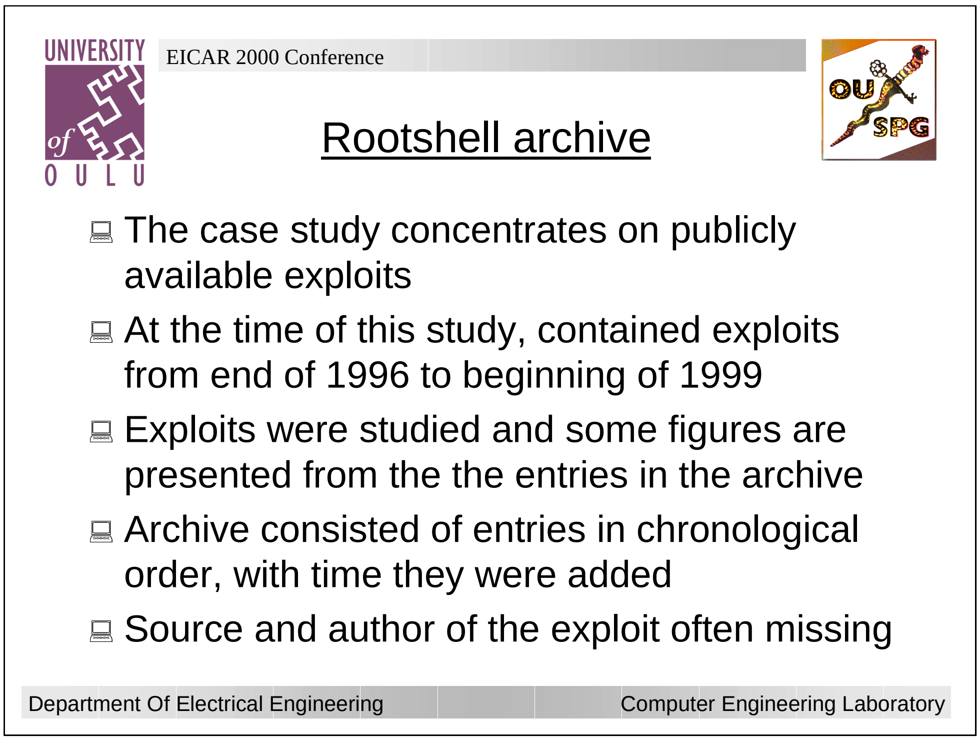
Channels for getting exploits

- Public disclosure mailing lists:
  - bugtraq@securityfocus.com
  - ntbugtraq@listserv.ntbugtraq.com
- Usenet newsgroups
- Collected in databases and web-pages
  - http://www.securityfocus.com
  - http://packetstorm.securify.com
  - http://www.rootshell.com

"Consumers don't care about security." - *from Bugtraq*

The best source of information on known vulnerabilities can be found from the archives of the public disclosure mailinglists and on some Usenet newsgroups.

Also good archives of existing exploits can be found. The paper and this presentation give examples drawn from the case study on exploits found from the Rootshell archive.


EICAR 2000 Conference

Rootshell archive

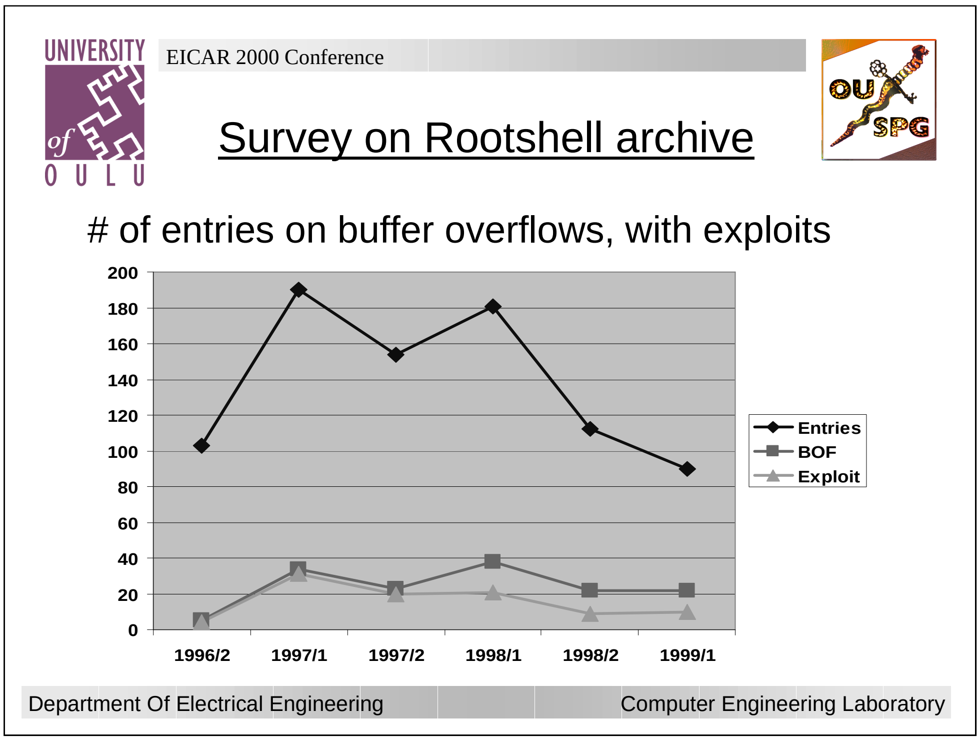
- The case study concentrates on publicly available exploits
- At the time of this study, contained exploits from end of 1996 to beginning of 1999
- Exploits were studied and some figures are presented from the the entries in the archive
- Archive consisted of entries in chronological order, with time they were added
- Source and author of the exploit often missing

Department Of Electrical Engineering Computer Engineering Laboratory

The case study consisted of exploits from years 1996 to 1999, and I will summarize some discoveries from this particular archive.

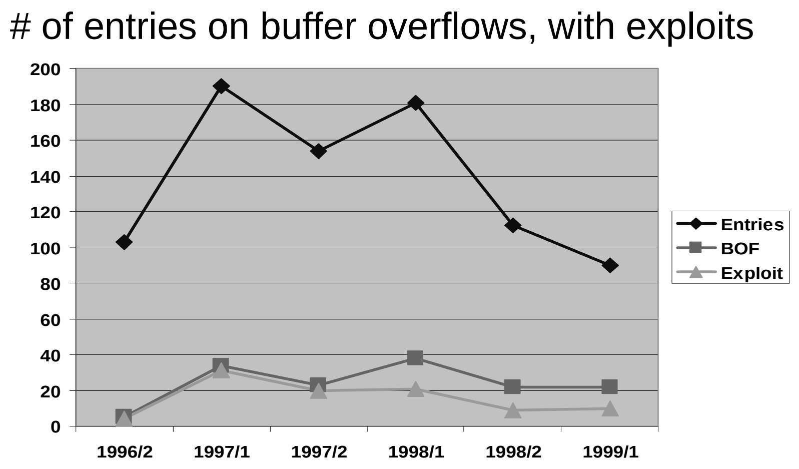
The major reason for choosing this archive was that it was arranged in chronological order, and was easy to browse through.

A drawback in the archive was that the original publication source and the author of the exploit itself was often missing.



EICAR 2000 Conference

Survey on Rootshell archive



Department Of Electrical Engineering Computer Engineering Laboratory

This chart summarizes the numbers of operational exploits, the attack recipes, in the Rootshell archive.

These are compared to the overall activity of the archive, the blue line.

The most active season for Rootshell was during 1997 and 1998.

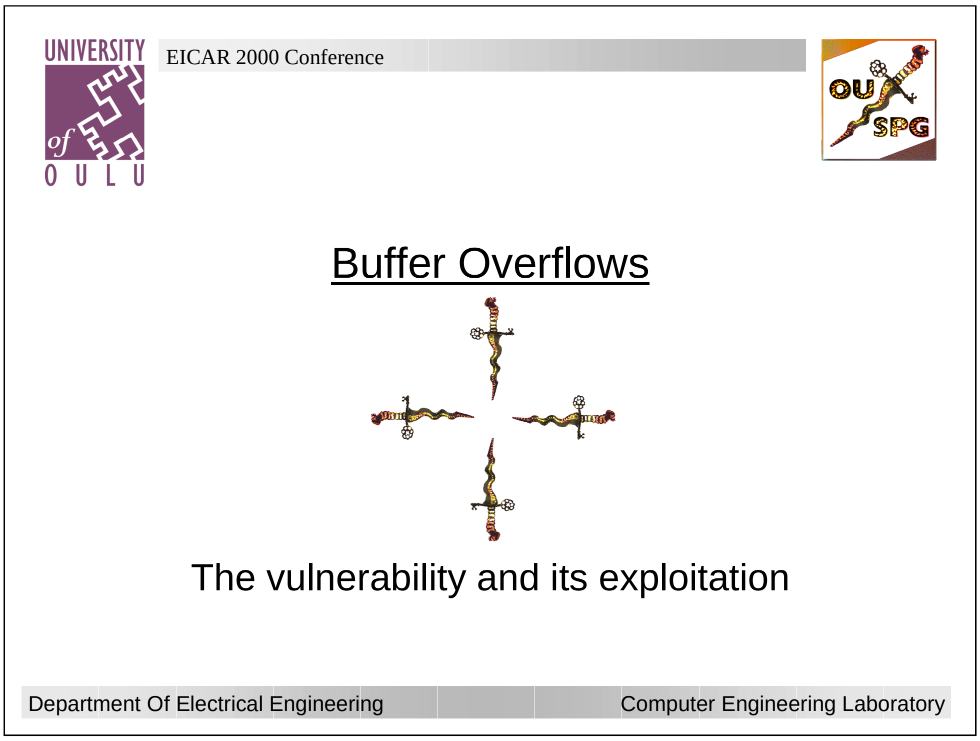
Most of the entries are about tools on hacking software or cracking passwords.

But, the number of entries on buffer overflow vulnerabilities, the red line, follows the general activity of the site.

Rootshell archive appears to have a boom right after most of the underground publications appeared.

Almost all of the entries for year 1997 contained an exploit, whereas only half for 1998 and 1999, as shown by the green line.

The buffer overflow is just announced and the generation of a working exploit is left for the public, or announced later.



EICAR 2000 Conference

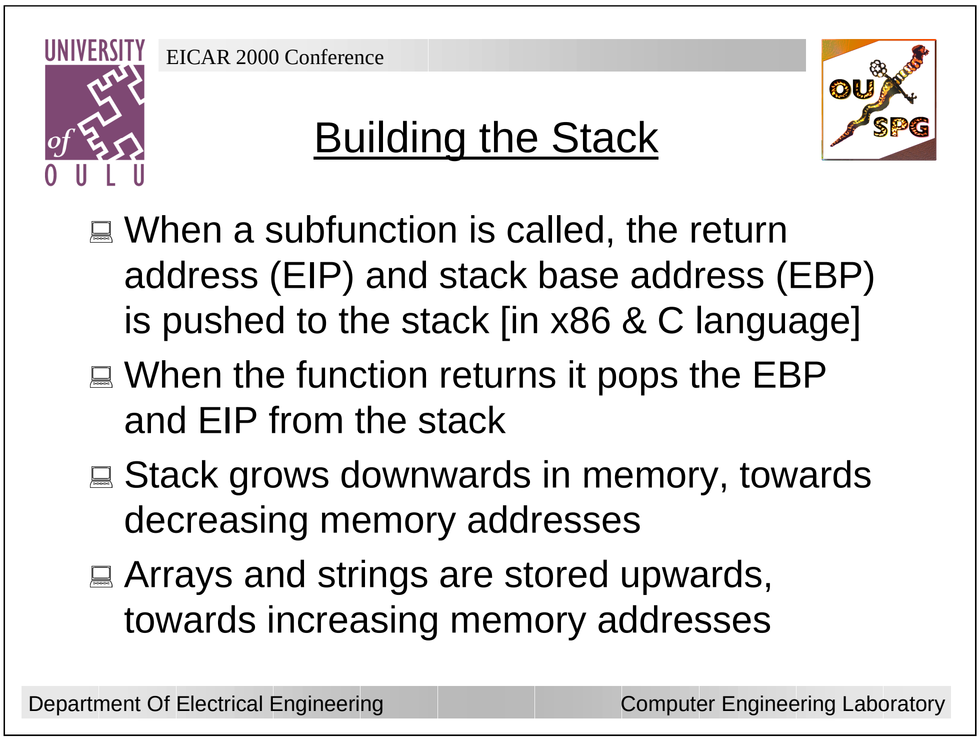
Buffer Overflows

The vulnerability and its exploitation

Department Of Electrical Engineering

Computer Engineering Laboratory

Lets go into more detail to the buffer overflows.



EICAR 2000 Conference

Building the Stack

- When a subfunction is called, the return address (EIP) and stack base address (EBP) is pushed to the stack [in x86 & C language]
- When the function returns it pops the EBP and EIP from the stack
- Stack grows downwards in memory, towards decreasing memory addresses
- Arrays and strings are stored upwards, towards increasing memory addresses

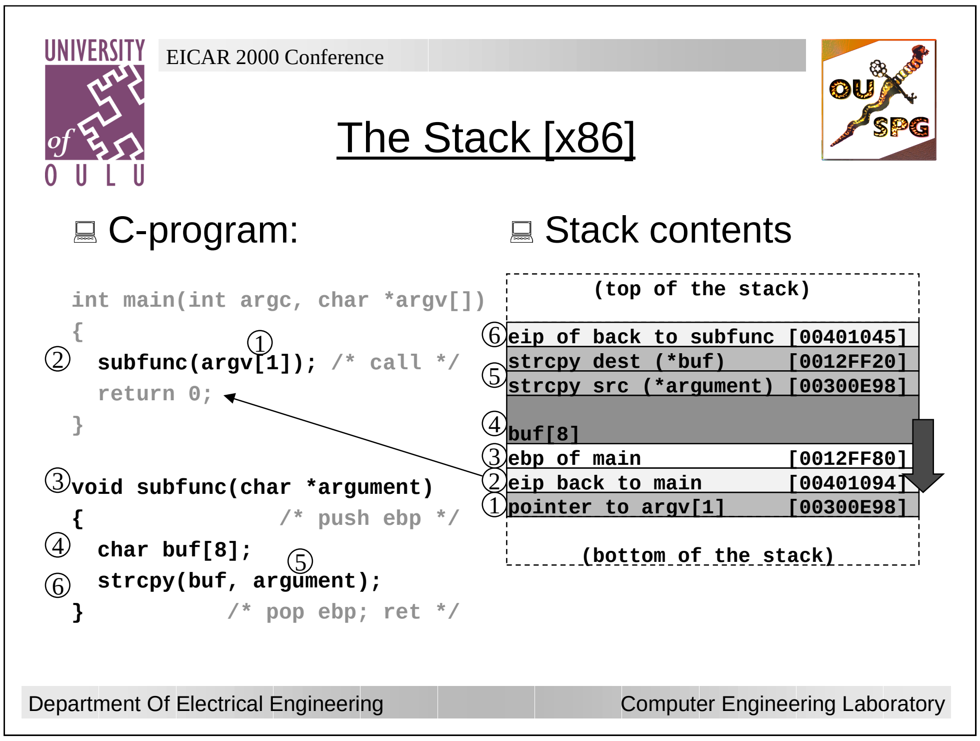
Department Of Electrical Engineering | Computer Engineering Laboratory

Next I will briefly present the operation of the stack frame thus causing the vulnerability.

Especially in C, when a subfunction is called, the current state of the thread is stored in the stack. The instruction pointer, called EIP in Intel, and the base pointer of the current stack-frame are both pushed into the stack.

When the function returns, they are popped back and the operation usually continues in the calling function.

When something is placed in the stack, the stack pointer value decreases, but the data stored in the stack is still referenced as in any other memory area.



EICAR 2000 Conference

The Stack [x86]

C-program:

```
  int main(int argc, char *argv[])
  {           ①
②   subfunc(argv[1]); /* call */
    return 0;
  }

③void subfunc(char *argument)
  {                /* push ebp */
④   char buf[8];     ⑤
⑥   strcpy(buf, argument);
  }            /* pop ebp; ret */
```

Stack contents

```
          (top of the stack)
⑥eip of back to subfunc [00401045]
 strcpy dest (*buf)     [0012FF20]
⑤strcpy src (*argument) [00300E98]
④
 buf[8]
③ebp of main            [0012FF80]
②eip back to main       [00401094]
①pointer to argv[1]     [00300E98]

         (bottom of the stack)
```

Department Of Electrical Engineering   Computer Engineering Laboratory

This slide shows a simple C-program that contains a subfunction calling an insecure string manipulation function. The operation is following:

1) A pointer or the value to the argument for the subfunction is pushed to the stack for usage in the subfunction.

2) When the subfunction is called, the return address is stored to the stack.

3) When the subfunction starts, the stack base pointer is usually pushed to the stack. The current stack pointer becomes the new base-pointer.

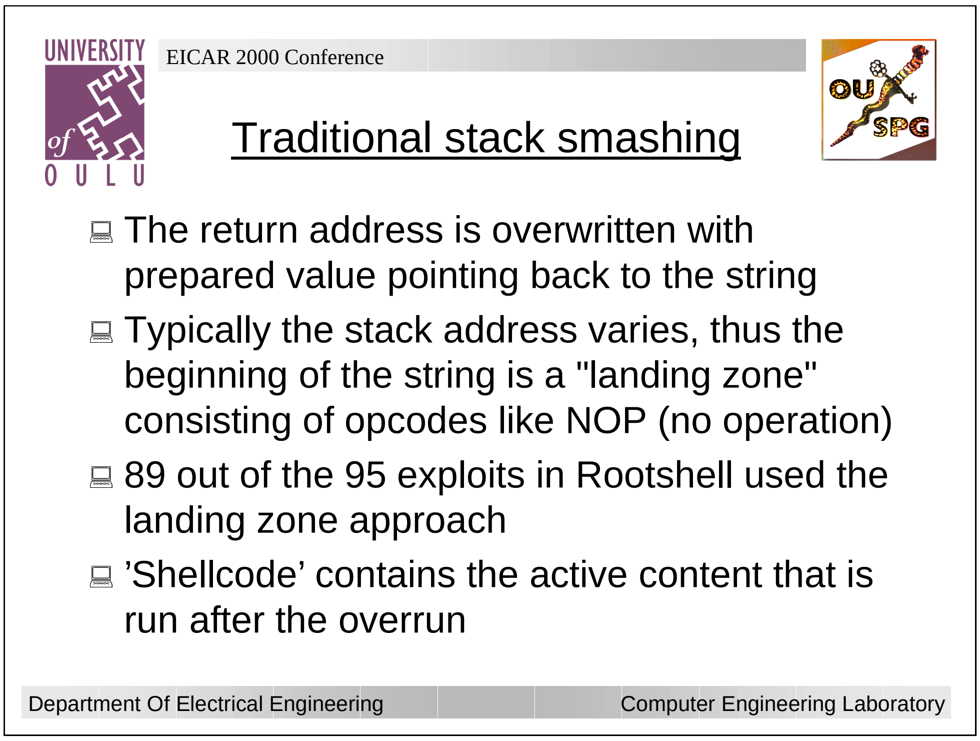
4) Space is reserved for local autovariables.

5) The arguments for the STRCPY function are pushed to the stack. In this case, the destination points to the reserved autovariable.

6) The function for string copy is called, and the return address pointing back to the subfunction is pushed to the stack.

In this example, the STRCPY-function returns nicely even if the buffer was overflowed. The red arrow shows the direction where the buffer overflows.

The problem starts when the subfunction returns and the return address is taken from the stack.


EICAR 2000 Conference

Traditional stack smashing

- The return address is overwritten with prepared value pointing back to the string
- Typically the stack address varies, thus the beginning of the string is a "landing zone" consisting of opcodes like NOP (no operation)
- 89 out of the 95 exploits in Rootshell used the landing zone approach
- ’Shellcode’ contains the active content that is run after the overrun

Department Of Electrical Engineering Computer Engineering Laboratory

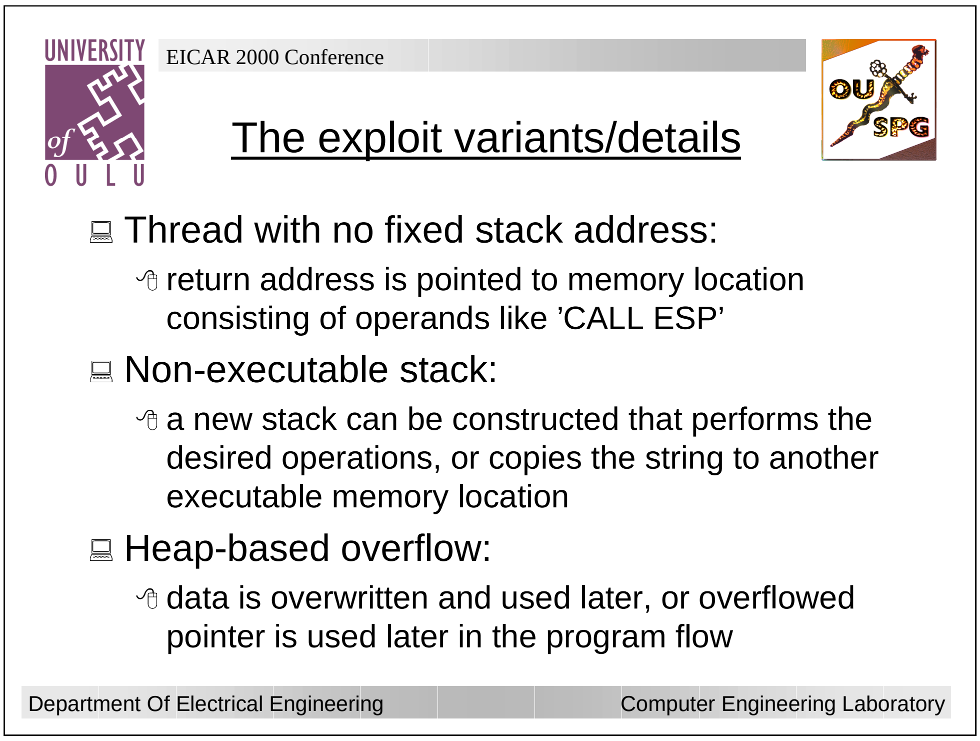
In a buffer overflow exploit, the string is formed up to contain a new return address that usually points right back to the known address inside the string itself.

Typically the address for the exploit can vary slightly and a ’landing zone’ or ’drop zone’ is required for the exploit to function. This can be created with assembly instructions like NOP, which stands for No Operation.

Most of the exploits available in the Rootshell archive used the landing zone approach for initiating the demonstration code, or the malicious code.

Followed by the landing zone is the shellcode, which is the actual active content of the exploit, and can consist of anything small enough to fit in the particular exploit.

This style of exploit can fail for many reasons. To overcome the difficulties, more advanced exploits have been presented.


EICAR 2000 Conference

The exploit variants/details

- Thread with no fixed stack address:
  - return address is pointed to memory location consisting of operands like ’CALL ESP’
- Non-executable stack:
  - a new stack can be constructed that performs the desired operations, or copies the string to another executable memory location
- Heap-based overflow:
  - data is overwritten and used later, or overflowed pointer is used later in the program flow

Department Of Electrical Engineering Computer Engineering Laboratory

If the stack address of the vulnerable thread varies a lot, a landing zone is not enough to start the exploit. But, when one searches the memory space of the process, one can easily find useful pieces of operands with fixed location. One good thing to look for is ’CALL ESP’ operands. This makes sure the exploit is started as the shellcode is in the stack.

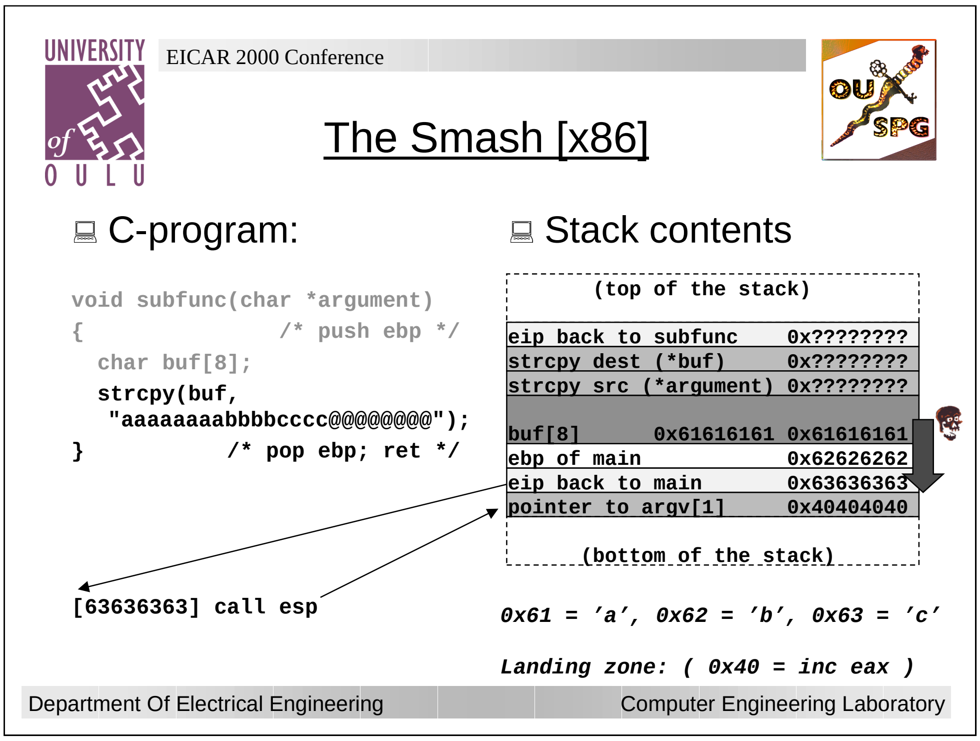
One example was available in the case study material.

Some operating systems can provide non-executable stack to protect against typical buffer overflow exploits. The protection can be avoided by constructing a valid stack-frame that calls the wanted system calls and provides parameters for them. Also, the shellcode can be run from an another memory location that is executable.

Three examples of this approach were available in the case study material.

Heap-based overflow exploits cannot use the return address for initiation, but rely on overwriting data or a pointer that is later used as a parameter or used for calling a subfunction.

One example was available in the case study material.



EIICAR 2000 Conference

The Smash [x86]

C-program:

```
void subfunc(char *argument)
{                /* push ebp */
  char buf[8];
  strcpy(buf,
   "aaaaaaaabbbbcccc@@@@@@@@");
}            /* pop ebp; ret */
```

Stack contents

```
        (top of the stack)

eip back to subfunc     0x????????
strcpy dest (*buf)      0x????????
strcpy src (*argument)  0x????????

buf[8]       0x61616161 0x61616161
ebp of main             0x62626262
eip back to main        0x63636363
pointer to argv[1]      0x40404040

      (bottom of the stack)
```

```
[63636363] call esp
```

0x61 = 'a', 0x62 = 'b', 0x63 = 'c'

Landing zone: (0x40 = inc eax)

Department Of Electrical Engineering   Computer Engineering Laboratory

In this little bit simplified image of the stack, we can see what happens when a buffer is overflowed.

In the C program, we note that a long string is copied to the buffer of 8 characters.

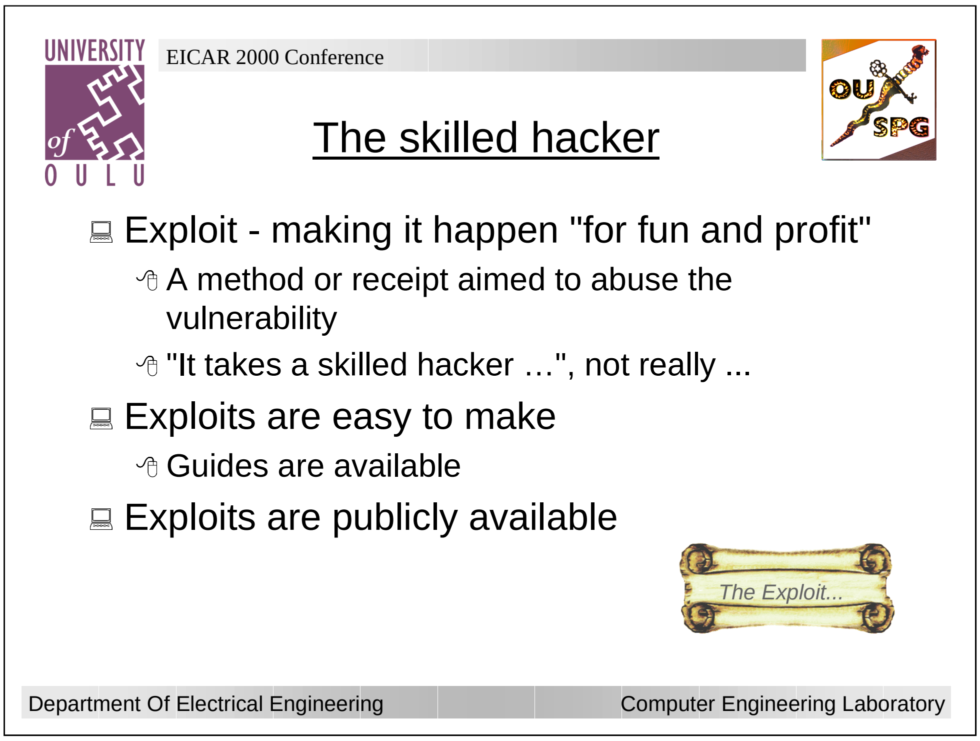
The red area shows the contents of the buffer, and we can see that the 8 first characters were stored there.

The red arrow shows the direction of growth of the string.

In this example, the next four letter 'b's overwrite the stored stack base pointer, and the next four letter 'c's overwrite the memory containing the return address back to the program flow.

Here, after returning from the subfunction, the program flow jumps to an address formed up by the letter 'c', which is address 0x63636363.

Now, if that address contains operands that perform a jump back to the stack, the program flow jumps back to the input string, which in this example contains operands 0x40, standing for increasing the value of the EAX register. This is a valid operand for using in a Landing Zone.



The skilled hacker

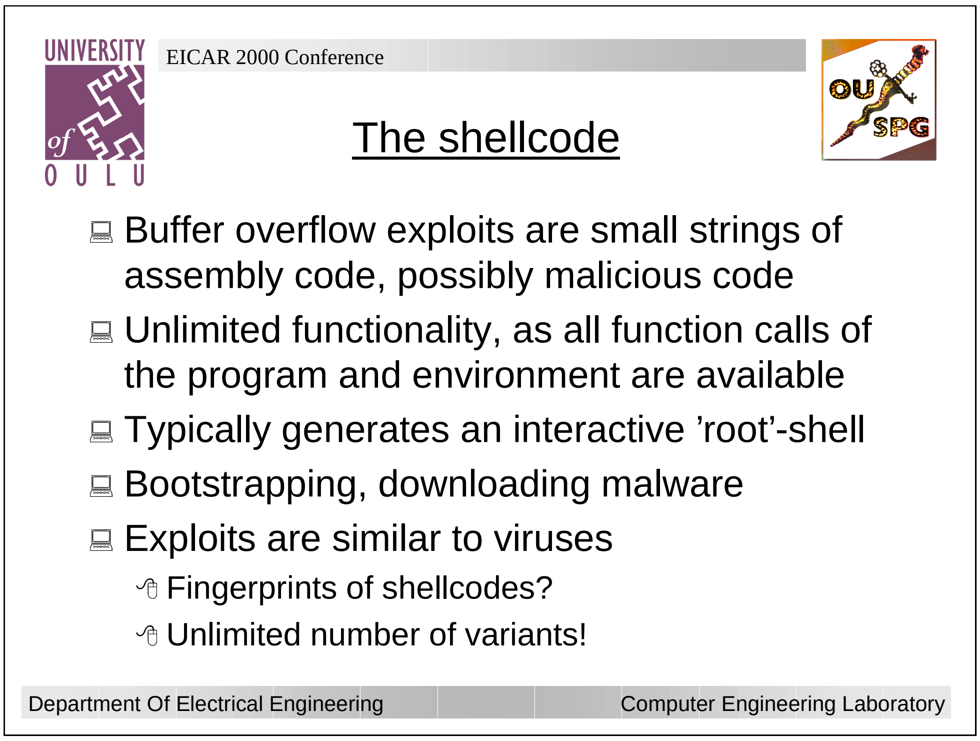
- Exploit - making it happen "for fun and profit"
  - A method or receipt aimed to abuse the vulnerability
  - "It takes a skilled hacker …", not really ...
- Exploits are easy to make
  - Guides are available
- Exploits are publicly available

The Exploit...

Typically the creation of exploits have been thought to be complex and the exploitation difficult, but it appears that all overflows are typically possible to exploit, and guides are easily available for generating an exploit.

When someone finally manages to create one for the vulnerability, the exploit is usually easy to find, and is widely distributed on open channels.

A typical name for a person that uses these prepared 'receipts' is a script-kiddie. Thus, one doesn't have to be a 'skilled hacker' to exploit a known vulnerability.


EICAR 2000 Conference

The shellcode

- Buffer overflow exploits are small strings of assembly code, possibly malicious code
- Unlimited functionality, as all function calls of the program and environment are available
- Typically generates an interactive 'root'-shell
- Bootstrapping, downloading malware
- Exploits are similar to viruses
  - Fingerprints of shellcodes?
  - Unlimited number of variants!

Department Of Electrical Engineering Computer Engineering Laboratory

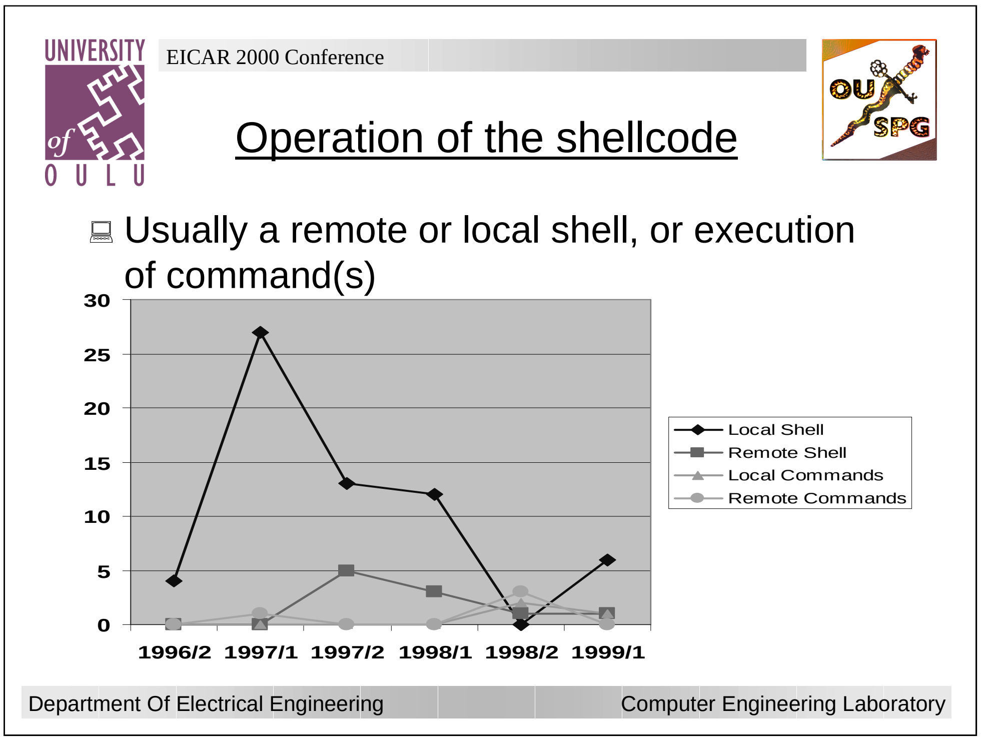
There is little limit on what one can do when exploiting a buffer overflow vulnerability.

All function calls that can be used in the vulnerable program can also be used in the exploit.

Any account achieved can then be used to get access to the machine, and there usually are ways of elevating privileges further.

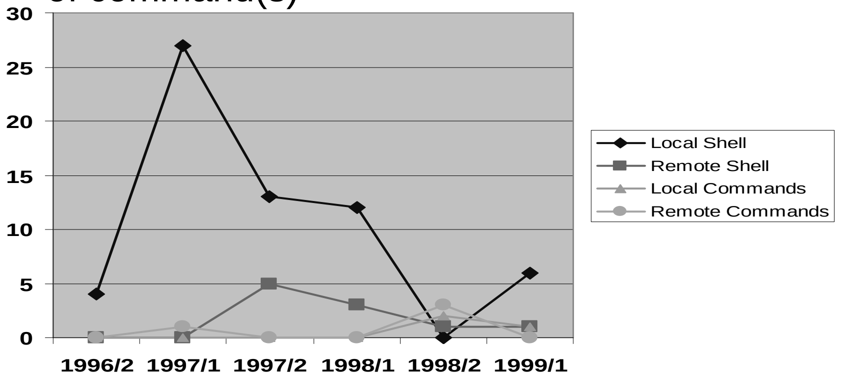
A possible method of increasing the functionality of the exploit is to generate shellcode that downloads and executes a prepared program, for example a trojan, on the victim's machine. One of the exploits in the case study material does that.

Virus scanners have fingerprints of viruses, why not overflow exploits? Some Intrusion Detection Systems, for example, perhaps could have fingerprints of popular exploits, but there can be unlimited number of variants of the exploits, and the variants are easy to create.



Operation of the shellcode

- Usually a remote or local shell, or execution of command(s)



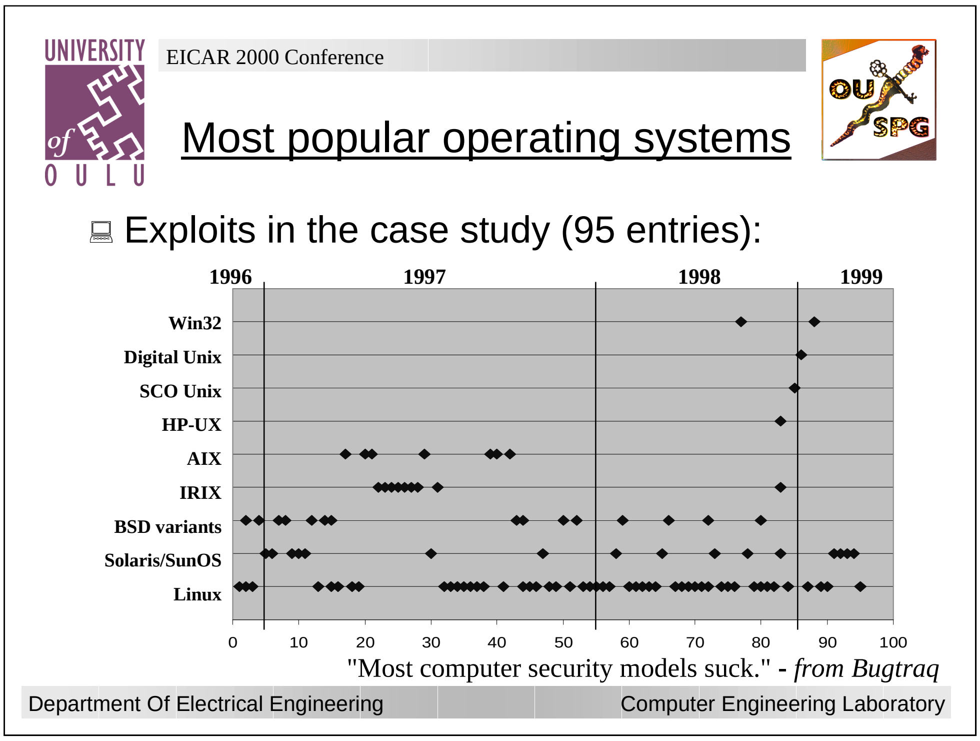
Here is some statistics on different operation found in the exploits in the case study material.

Most typical shellcode just provides a local or remote shell, and some exploits just execute commands on the remote host. The operation usually depends on the vulnerable program.

There is a trend of going from simple 'local shell' exploits into more sophisticated shellcode with more advanced operation.

More advanced payload for the exploits found in Rootshell include: remote x-terminal, download and execute, changing permissions on files, creating files, changing the hostname, reboot, modifying password file and creating setuid shell.

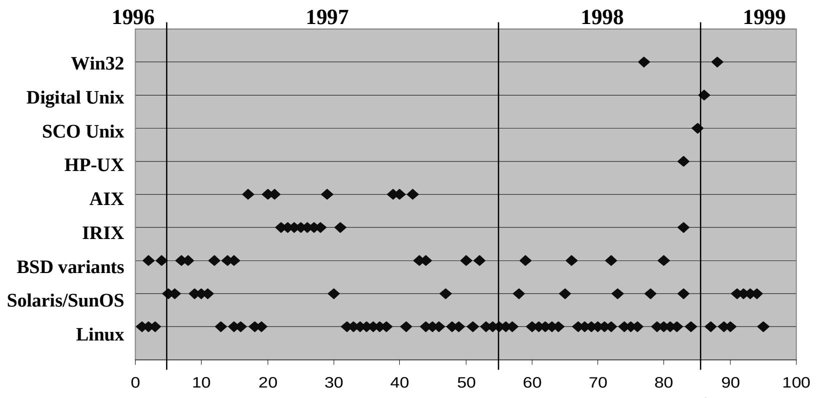
There is no limit what the shellcode can do within the privileges of the vulnerable process.



EICAR 2000 Conference

Most popular operating systems

- Exploits in the case study (95 entries):



"Most computer security models suck." - *from Bugtraq*

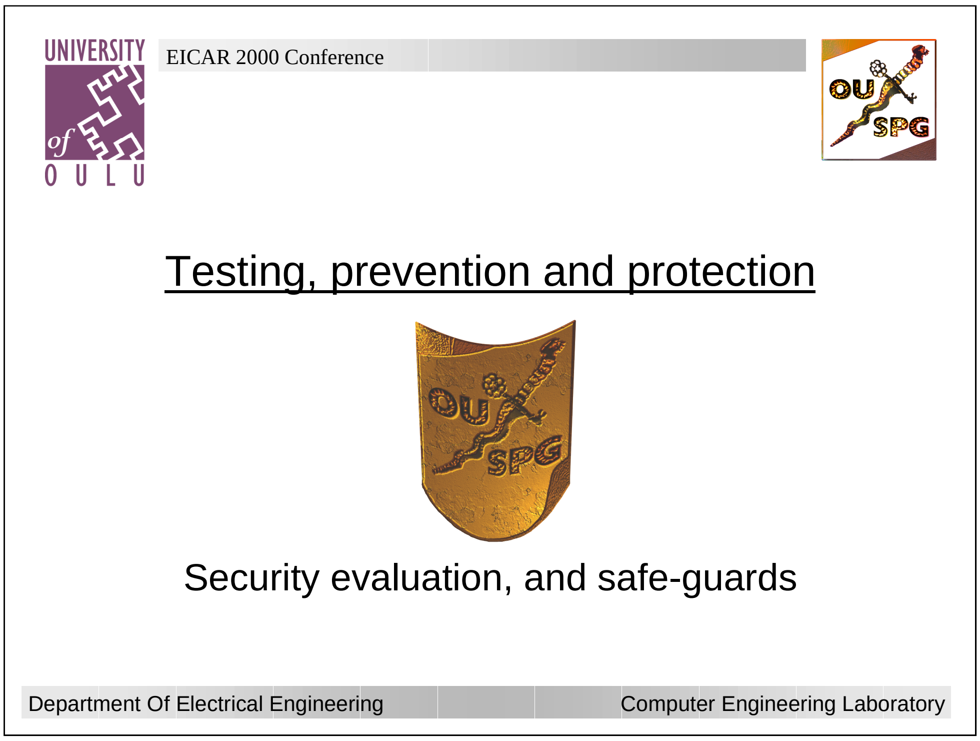
Department Of Electrical Engineering   Computer Engineering Laboratory

In this chart we can see the distribution of exploits between operating systems.

Linux, Solaris and BSD variants can be noted to have been the favorite platforms for the people discovering overflows and making the exploits against them.

The availability of an operating system for home use, and the availability of the source code probably affect the numbers most. Still, lately, all popular operating systems have had their share of these faults.

As seen here, the popularity of exploits against Windows operating systems and against Macintoshes is really low. Also a note about a buffer overflow on Macintosh was found on the archive studied, but the entry did not contain an operational exploit.



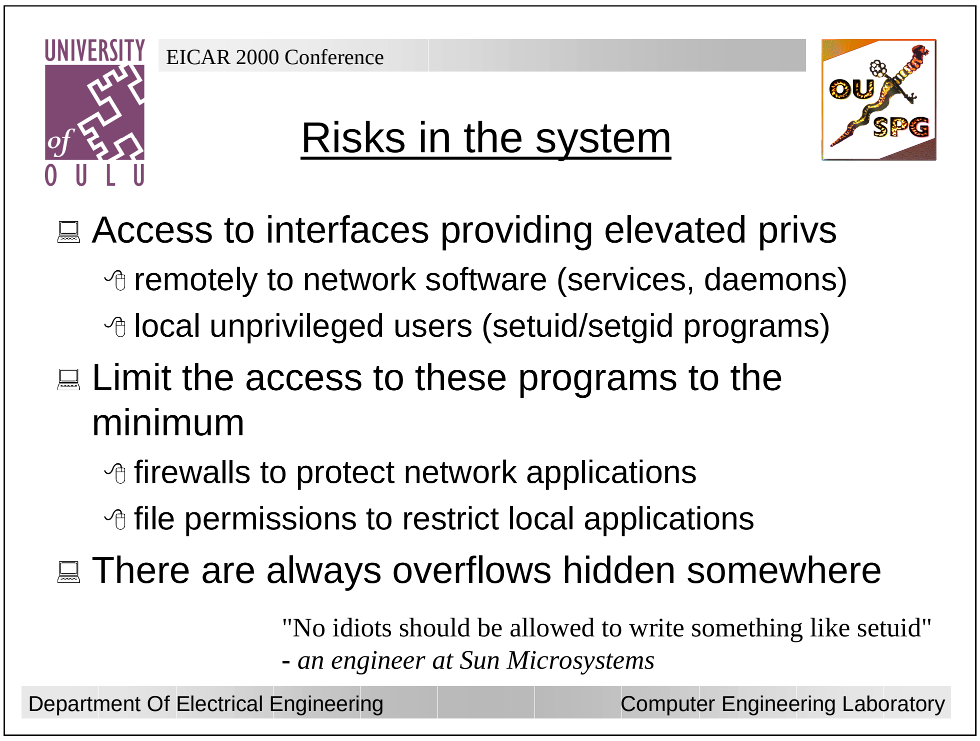
EICAR 2000 Conference

Testing, prevention and protection

Security evaluation, and safe-guards

Department Of Electrical Engineering Computer Engineering Laboratory

I will briefly discuss some existing methods for discovering these faults proactively, and protecting the system from them, reactively.



Risks in the system

- Access to interfaces providing elevated privs
  - remotely to network software (services, daemons)
  - local unprivileged users (setuid/setgid programs)
- Limit the access to these programs to the minimum
  - firewalls to protect network applications
  - file permissions to restrict local applications
- There are always overflows hidden somewhere

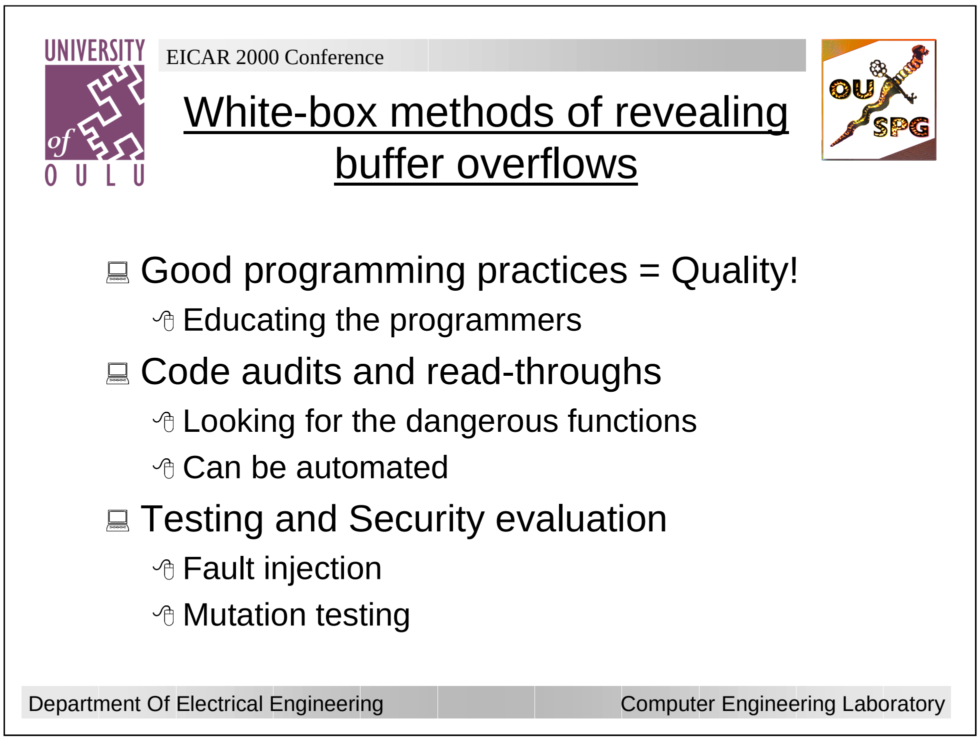
"No idiots should be allowed to write something like setuid"
- *an engineer at Sun Microsystems*

When we consider looking for this kind of errors, the attention should first be turned on interfaces to the network, and to programs operating with high privileges, and that can be used by non-privileged people

There are simple solutions of restricting access to these programs both from the network and from the filesystem.

Typically, there always are some overflows hidden somewhere waiting to be found, and it can be difficult to decide which software to trust and rely on.

Some user installable software such as such as email clients and web browsers can also contain overflows.



White-box methods of revealing buffer overflows

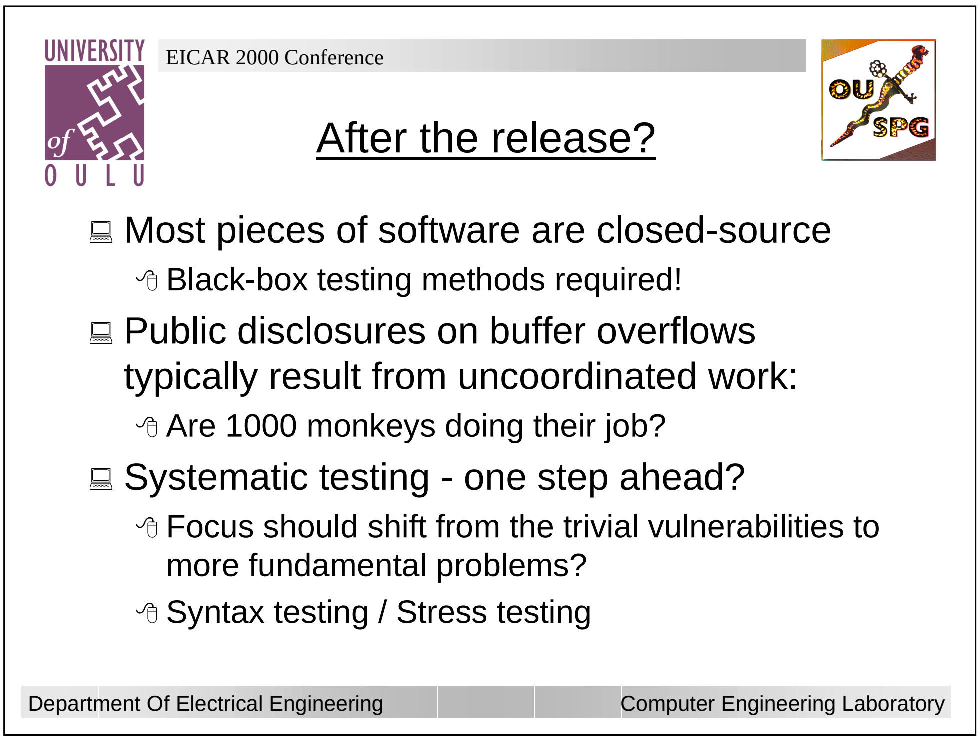
- Good programming practices = Quality!
  - Educating the programmers
- Code audits and read-throughs
  - Looking for the dangerous functions
  - Can be automated
- Testing and Security evaluation
  - Fault injection
  - Mutation testing

The only place to fix these faults is in the development of the software.

Some quality in should be required from the developers.

This includes methods of evaluating the source code before the release, and thorough testing for this type of security faults.

Some white box testing methods have attempted to discover these faults.



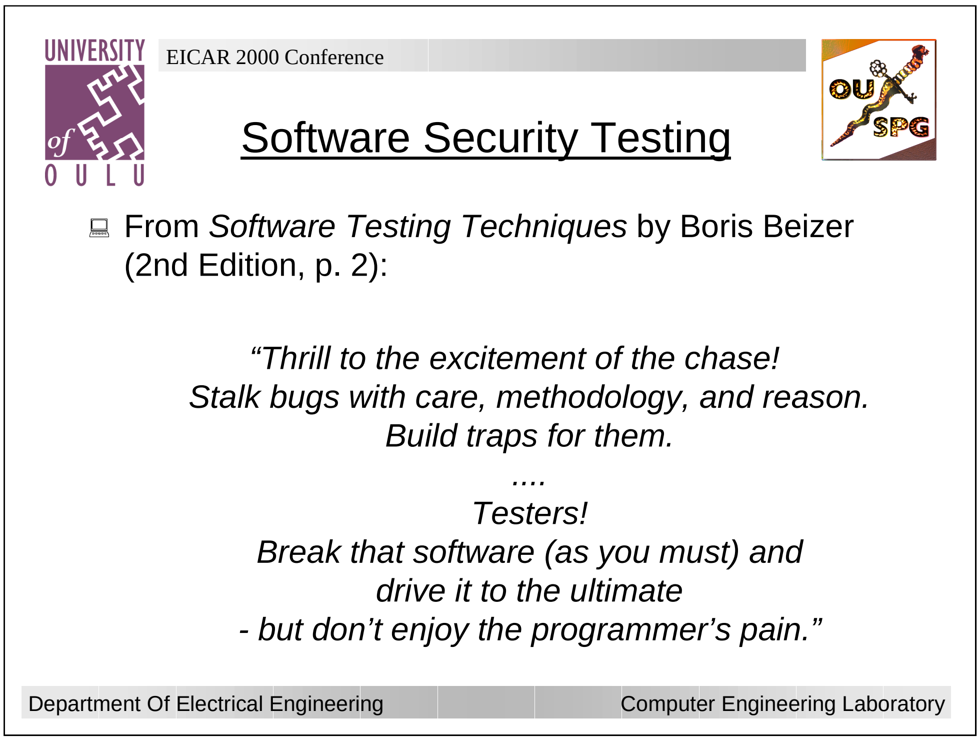
After the release?

- Most pieces of software are closed-source
  - Black-box testing methods required!
- Public disclosures on buffer overflows typically result from uncoordinated work:
  - Are 1000 monkeys doing their job?
- Systematic testing - one step ahead?
  - Focus should shift from the trivial vulnerabilities to more fundamental problems?
  - Syntax testing / Stress testing

The evaluation of closed-source software after the release have been considered by our group. This requires black box testing methods.

Generally most of the public disclosures on emaillists can be seen as non-systematic, chaotic engine of black box testing. These people can be seen as thousand monkeys trying out all possible inputs to the software.

A more systematic, and proactive methods should be available for the developers. At least simple programming errors such as the overflows are, can be discovered with testing methods like syntax testing and stress testing.



Software Security Testing

- From *Software Testing Techniques* by Boris Beizer (2nd Edition, p. 2):

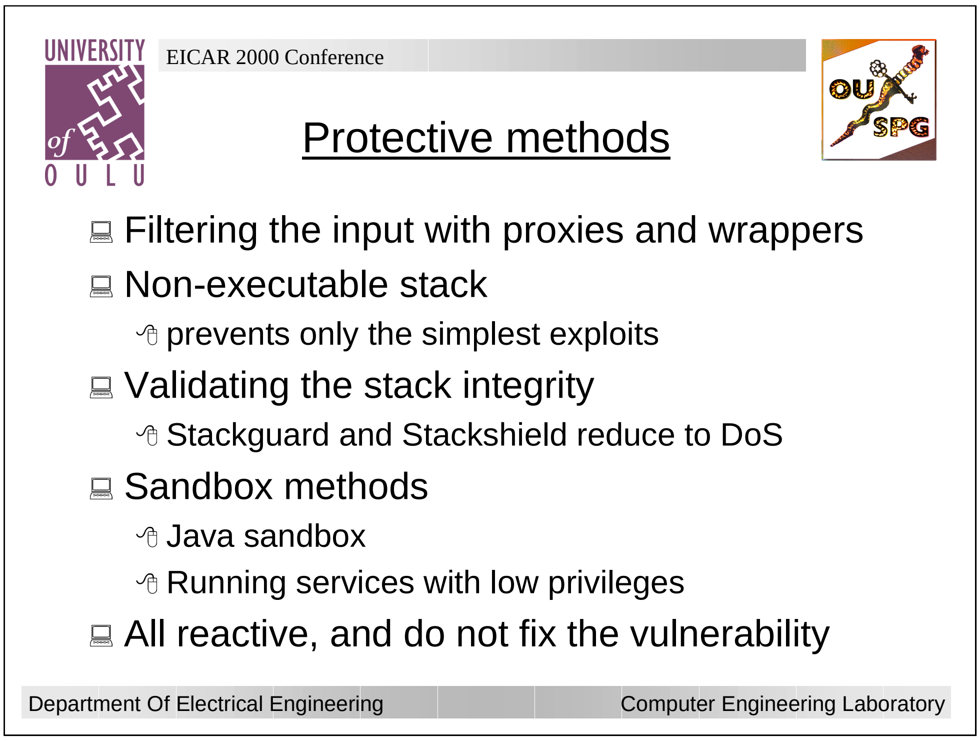
“Thrill to the excitement of the chase!
Stalk bugs with care, methodology, and reason.
Build traps for them.
....
Testers!
Break that software (as you must) and
drive it to the ultimate
- but don’t enjoy the programmer’s pain.”

Boris Beizer has worded out just what we we think about testing:

It is exciting!

Breaking software is fun and interesting!

Still, we have to be merciful to the developers, and have a professional approach to the testing.



Protective methods

- Filtering the input with proxies and wrappers
- Non-executable stack
  - prevents only the simplest exploits
- Validating the stack integrity
  - Stackguard and Stackshield reduce to DoS
- Sandbox methods
  - Java sandbox
  - Running services with low privileges
- All reactive, and do not fix the vulnerability

Reactive protective methods limit the exploitation somewhat, but there usually exist work-arounds enabling the exploitation.

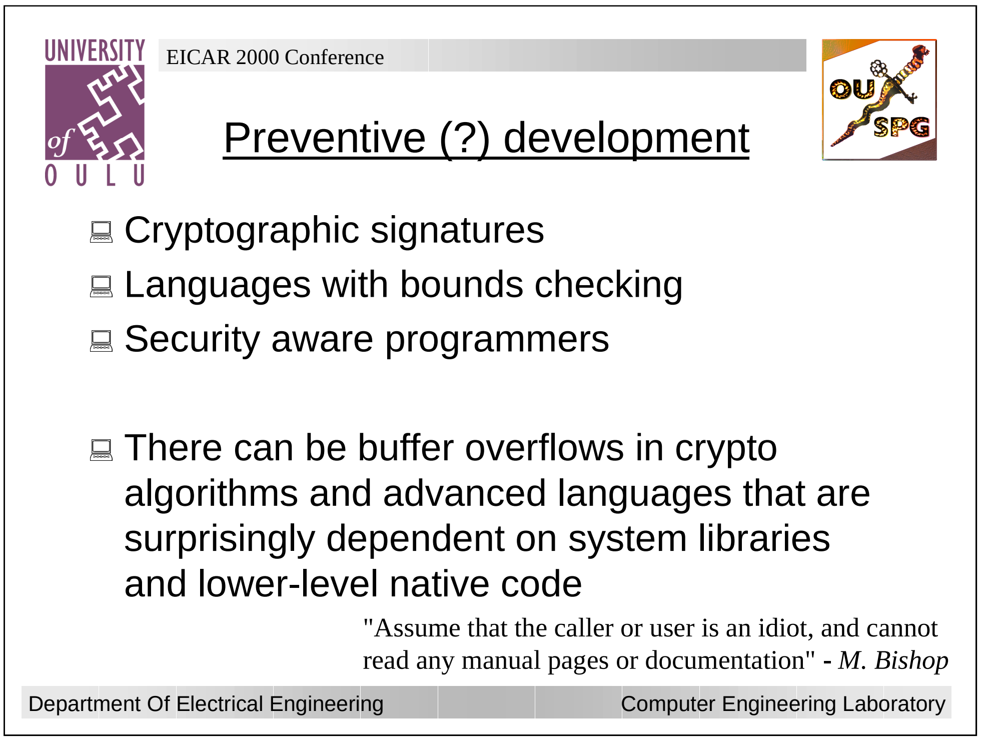
In the case study, there were several examples of how the input filtering could be worked around by encoding the code into something the victim will allow. Instructions for creating a working exploit that uses only alphanumeric characters were available in the case study material.

Operating system modifications, such as the non-executable stack prevent the execution of shellcode from the stack. Several exploitation methods were available that worked around the non-executable stack protection.

Compiler modifications such as Stackguard and Stackshield use different approaches in validating the stack integrity before returning the control to the calling function. At best, these protection methods reduce the exploit into a Denial of Service attacks.

Sandbox methods reduce the damage after the possible penetration. If a sandboxed application is successfully exploited, the attacker can start looking for a route outside the box.

None of these prevent the actual fault, just reduce the impact.



EICAR 2000 Conference

Preventive (?) development

- Cryptographic signatures
- Languages with bounds checking
- Security aware programmers

- There can be buffer overflows in crypto algorithms and advanced languages that are surprisingly dependent on system libraries and lower-level native code

"Assume that the caller or user is an idiot, and cannot read any manual pages or documentation" - *M. Bishop*

Department Of Electrical Engineering Computer Engineering Laboratory

Perhaps there are methods of preventing these kind of attacks or the development of the programming errors?

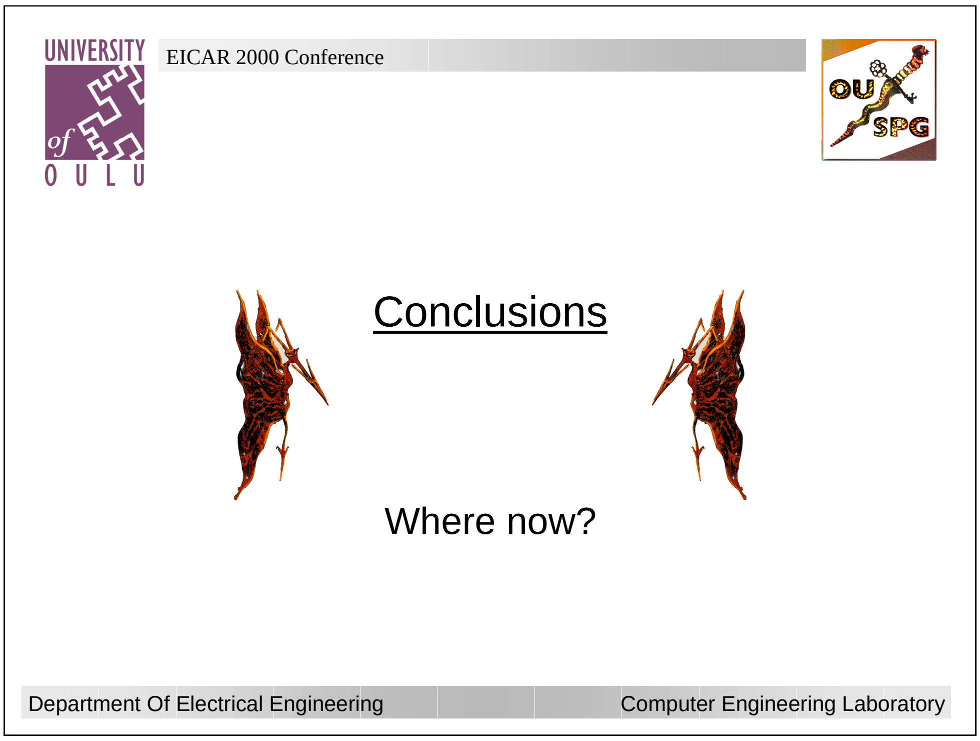
Cryptographic signatures limit the perpetrators to someone that can be verified.

Languages with bounds checking will not have these errors.

Ultimately, skilled and security aware programmers are the best solution. Security should be taken seriously by software developers and there is never enough education on good programming practices.

At this time, this is still something that is improving. Buffer overflows in crypto software and low level system libraries can still introduce errors in current solutions.

Security aware programmers are scarce, there are never enough skilled programmers around.



EICAR 2000 Conference

Conclusions

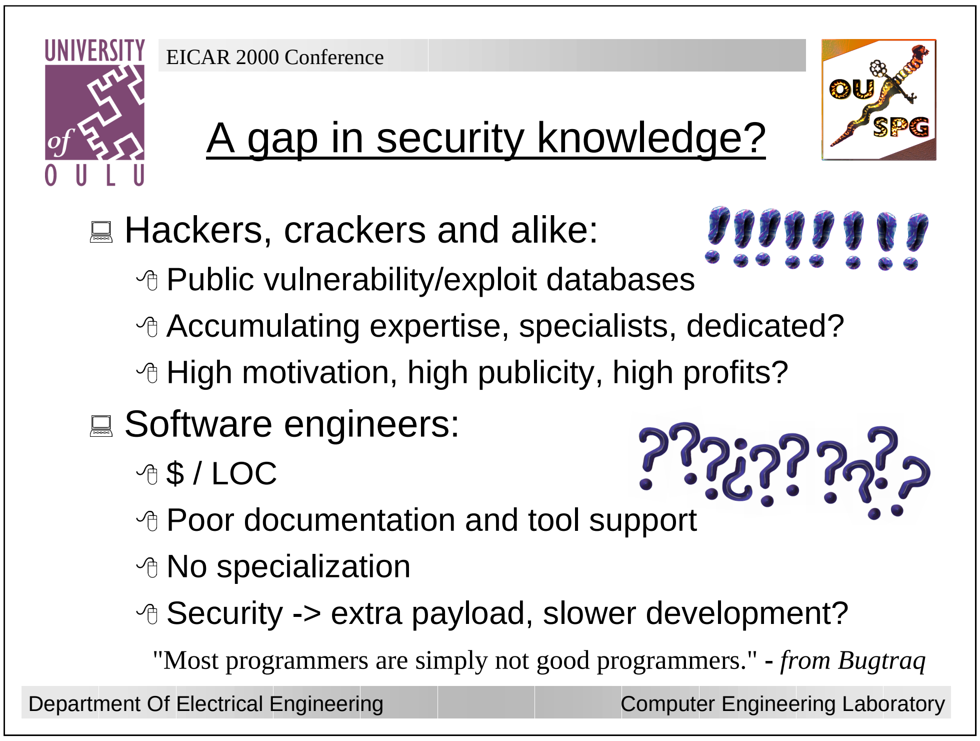
Where now?

Department Of Electrical Engineering

Computer Engineering Laboratory

Few concluding slides.

Where are we now and where are we going?



EICAR 2000 Conference

A gap in security knowledge?

- Hackers, crackers and alike:
  - Public vulnerability/exploit databases
  - Accumulating expertise, specialists, dedicated?
  - High motivation, high publicity, high profits?
- Software engineers:
  - $ / LOC
  - Poor documentation and tool support
  - No specialization
  - Security -> extra payload, slower development?

"Most programmers are simply not good programmers." - *from Bugtraq*

Department Of Electrical Engineering   Computer Engineering Laboratory

Who are the experts of this field?

People who have taken security in their life maintain public exploit databases.

They are dedicated to their field and follow the development around the globe.

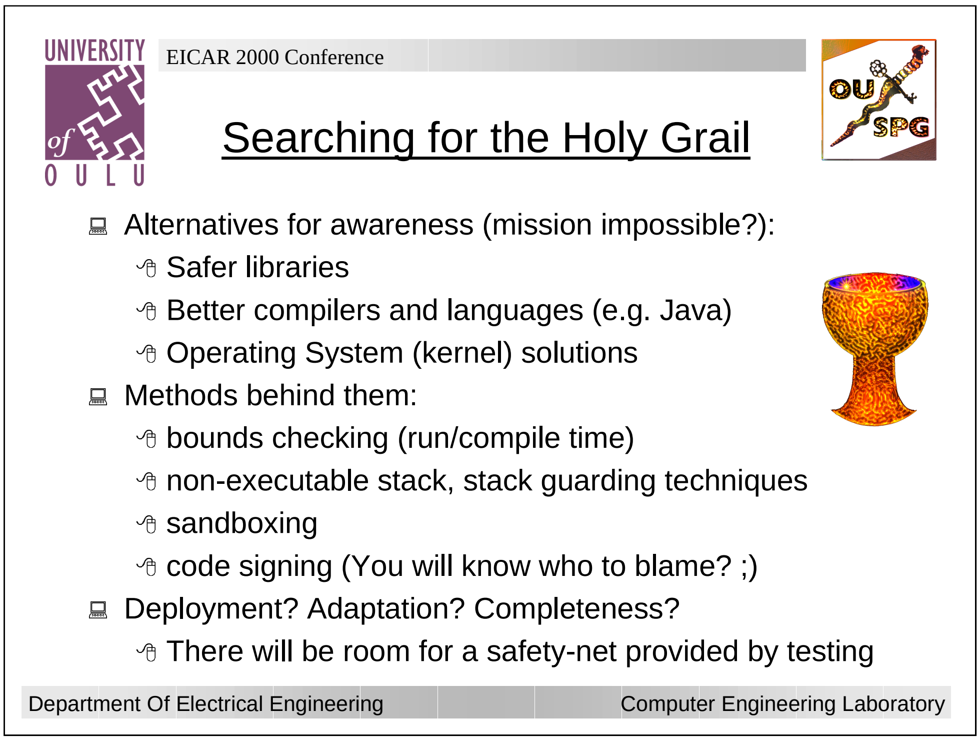
These hacker-type people are highly motivated, and some groups have gained high publicity, and even profit with their skills.

On the other hand, the working people fighting with the actual code usually work to produce code, not secure products.

Security products usually are unusable by people that do not have time nor motivation to use them.

If someone happens to learn something from the security field, or shows himself to be a ’good programmer’, he is quickly promoted to somewhere he cannot do any good.

For software development, security is a hindrance, and just slows down the production.



EICAR 2000 Conference

Searching for the Holy Grail

- Alternatives for awareness (mission impossible?):
  - Safer libraries
  - Better compilers and languages (e.g. Java)
  - Operating System (kernel) solutions
- Methods behind them:
  - bounds checking (run/compile time)
  - non-executable stack, stack guarding techniques
  - sandboxing
  - code signing (You will know who to blame? ;)
- Deployment? Adaptation? Completeness?
  - There will be room for a safety-net provided by testing

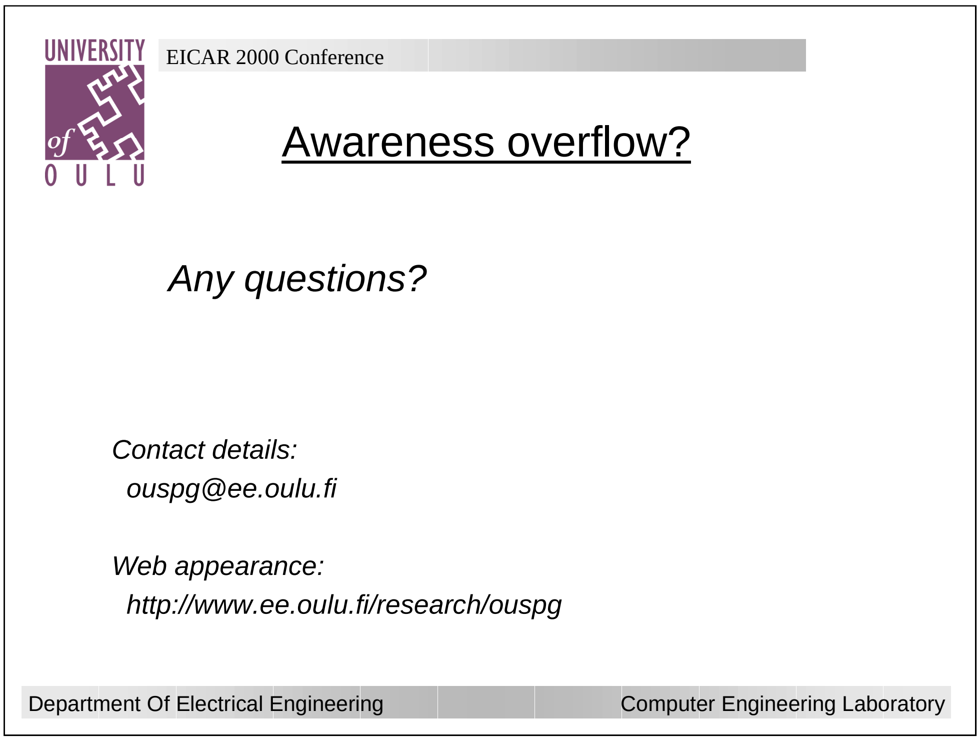
Department Of Electrical Engineering Computer Engineering Laboratory

If we cannot have good programmers, then maybe we have to deploy the available solutions. Safer libraries, whole new languages and operating system solutions come to the aid.

These have been made possible with bounds checking properties, non-executable memory areas and different stack properties. Sandboxes and code signing give some restriction on what the attacker can do, and who can attack at all.

Still, current systems are far from clean from buffer overflows, and all these presented methods have weaknesses.

There is always work in developing improved testing methods.



Awareness overflow?

Any questions?

Contact details:
ouspg@ee.oulu.fi

Web appearance:
http://www.ee.oulu.fi/research/ouspg

I hope this wasn't an awareness overflow!

References to some good sources of additional information can be found from the accompanying paper.

Here is our email address, and we will also provide some additional information in our homepage.

If you have questions, please ask!