





Welcome to

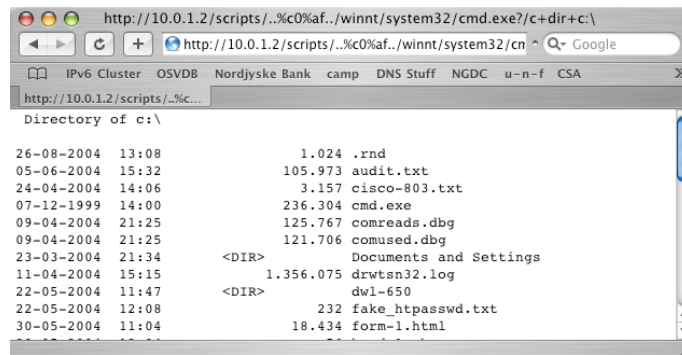
## 9. Strings and Metacharacters

KEA Competence OB2 Software Security

Henrik Kramselund he/him han/ham hlk@zencurity.com @kramse  

Slides are available as PDF, kramse@Github  
9-strings-and-metacharacters.tex in the repo security-courses

## Goals:



Strings are used in most programs,

We have seen problems with strings for decades like Microsoft IIS 4.0/5.0 Unicode bug CVE-2000-0884

Handling letters, numbers, sentences, filenames, ... - string data

Multiple data formats, from American Standard Code for Information Interchange (ASCII), Extended Binary Coded Decimal Interchange Code (EBCDIC), ISO 8859-1 / ISO-8859-15 EURO sign

€

From 7-bit ASCII, 8-bit ASCII to multibyte symbols in Unicode

Lots of opportunity for errors, searching on google for *unicode bug CVE* gave 500.000 hits!



# Plan for today



## Subjects

- Processing strings
- C String handling
- Metacharacters
- Character sets and unicode

## Exercises

- Recommendations for handling strings, how does Python help, how does Django handle strings, and input validation
- Truncate and Encoding Attacks JuiceShop
- Django String Handling

## Reading Summary



*Gray Hat Hacking* chapters 1-2, 11-13 - browse if you need to, many pages.

Browse: *Return-Oriented Programming: Systems, Languages, and Applications* and *Removing ROP Gadgets from OpenBSD*

Also checkout [https://en.wikipedia.org/wiki/C\\_string\\_handling](https://en.wikipedia.org/wiki/C_string_handling) for use when you dont have the books with you.

Wikipedia calls format string vulnerabilities Uncontrolled format string [https://en.wikipedia.org/wiki/Uncontrolled\\_format\\_string](https://en.wikipedia.org/wiki/Uncontrolled_format_string)

# Processing strings



Many of the most significant security vulnerabilities of the last decade, (1997-2007) are the result of memory corruption due to mishandling textual data, or logical flaws due to the misinterpretation of the content on the textual data

Source: *The Art of Software Security Assessment Identifying and Preventing Software Vulnerabilities* 2007

Spoiler, the problems didn't end in 2007

Major areas of string handling:

- memory corruption due to string mishandling
- Vulnerabilities due to in-band control data in the form of metacharacters
- Vulnerabilities resulting from conversions between character encodings in different languages

By understanding the **common patterns** associated with these vulnerabilities, you can identify and prevent their occurrence

# C String handling



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[10];
    strcpy(buf, argv[1]);
    printf("%s\n",buf);
}
```

- In C there is no native type for strings; strings are formed by constructing arrays of the char data type, with the null character (0x00) marking the end of a string
- The string is then represented by the pointer to the beginning, buf
- C++ standard library has a string class, a little safer
- Converting between C++ string class and C strings may result in vulnerabilities
- Many systems use C at the bottom, C APIs etc.

# Unbounded String Functions



Unsafe group of functions:

- **scanf()** read data from somewhere, multiple variants
- **sprintf()** print formatted into string/buffer - overflow  
Changing the format string is a whole group in itself
- **strcpy()** family is notorious for causing a large number of security vulnerabilities
- **strcat()** string concatenation, combining strings can be problematic

They will continue whatever they do, until they meet a null-terminator or get an error, segmentation faults

These were the ones people used in the beginning, and forever



# Many Years ago around 1988



```
/usr/src/etc/fingerd.c from 4.3BSD:
main(argc, argv)
    char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    ...
    line[0] = '\0';
    gets(line);
}
```

Source code link <https://www.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c>

More description in the articles:

<https://spaf.cerias.purdue.edu/tech-reps/823.pdf> *The Internet Worm Program: An Analysis* Purdue Technical Report CSD-TR-823

Eugene H. Spafford

<https://blog.rapid7.com/2019/01/02/the-ghost-of-exploits-past-a-deep-dive-into-the-morris-worm/>

The Ghost of Exploits Past: A Deep Dive into the Morris Worm

# Exim CVE-2019-15846 git diff exim-4.92.1 exim-4.92.2



```
diff --git a/src/src/string.c b/src/src/string.c
```

```
@@ -224,6 +224,8 @@ interpreted in strings.
```

Arguments:

```
pp      points a pointer to the initiating \ in the string;
        the pointer gets updated to point to the final character
+       If the backslash is the last character in the string, it
+       is not interpreted.
```

Returns: the value of the character escape

```
*/
```

```
@@ -236,6 +238,7 @@ const uschar *hex_digits= CUS"0123456789abcdef";
```

```
int ch;
const uschar *p = *pp;
ch = *(++p);
+if (ch == '\\0') return **pp;
```

The vulnerability is exploitable by sending a SNI ending in a backslash-null sequence during the initial TLS handshake. The exploit exists as a POC. More details see `doc/doc-txt/cve-2019-15846/` in the source code repository.

Internet worm 1988 – Exim 2019, about 30 years of *progress*

## Printf – formatted printing



The `printf()` function can be used to print more than just fixed strings. This function can also use format strings to print variables in many different formats. A format string is just a character string with special escape sequences that tell the function to insert variables printed in a specific format in place of the escape sequence. The way the `printf()` function has been used in the previous programs, the `"Hello, world!\n"` string technically is the format string; however, it is devoid of special escape sequences. These escape sequences are also called format parameters, and for each one found in the format string, the function is expected to take an additional argument. Each format parameter begins with a percent sign (similar to formatting characters used by GDB's `examine` command).

Source: *Hacking, 2nd Edition: The Art of Exploitation*, Jon Erickson

## Examples printf



```
// Example of printing with different format string
printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
printf("[string] %s Address %08x\n", string, string);
// Example of unary address operator (dereferencing) and a %x format string
printf("variable A is at address: %08x\n", &A);
```

Can also print into another string (buffers) with functions like `sprintf()` – string printf

Source: *Hacking, 2nd Edition: The Art of Exploitation*, Jon Erickson



## Exploiting format string vulnerabilities

A format string exploit is another technique you can use to gain control of a privileged program. Like buffer overflow exploits, format string exploits also depend on programming mistakes that may not appear to have an obvious impact on security. Luckily for programmers, once the technique is known, it's fairly easy to spot format string vulnerabilities and eliminate them. Although format string vulnerabilities aren't very common anymore, the following techniques can also be used in other situations.

### Summary:

- 0x353 Reading from Arbitrary Memory Addresses
- 0x354 Writing to Arbitrary Memory Addresses
- Plus some tips and hints for making it easier

# Bounded String Functions



Adding a maximum length to the functions should help:

- **snprintf()** copies a maximum number of bytes!
- Different semantics on Windows and Unix.
- Windows does not guarantee null-termination, returns -1
- Unix guarantee null-termination, returns number of chars that would have been written had there been enough room
- **strncpy()** does accept a maximum number of bytes to be copied into the destination, but does not guarantee null termination
- **strncat()** size to provide is the space left in the buffer, not the size of the whole buffer
- Easy to result in off-by-one vulnerabilities

Programming is easy, right 😊

# Better Functions from BSD



- strncpy, strlcat size-bounded string copying and concatenation
- **strncpy()** a variant of strcpy that truncates the result to fit in the destination buffer
- **strlcat()** a variant of strcat that truncates the result to fit in the destination buffer
- Originally OpenBSD 2.4 in December, 1998
- These functions always write one null to the destination buffer
- May truncate the result, return size of buffer needed, programmer must check return code and handle this

Example references:

[https://en.wikipedia.org/wiki/C\\_string\\_handling](https://en.wikipedia.org/wiki/C_string_handling)

[https://en.wikibooks.org/wiki/C\\_Programming/C\\_Reference/nonstandard/strncpy](https://en.wikibooks.org/wiki/C_Programming/C_Reference/nonstandard/strncpy)

# Parsing String Data



```
char t[1000];  
char tt[100];
```

```
// Get data into t from somewhere
```

```
while (*t != ':') {  
    *tt++ = *t++;  
}  
*tt = 0;
```

- Example, if the input is larger than destination pointed to by `tt` then problems can arise
- Character expansion, making output bigger can overflow
- Other examples can be found across the internet over 30+ years



# Metacharacters



- Null 0x00, special in C, but just another char in other languages
- Space
- / used as filename delimiters, and \ in Windows
- . dot used in various ways for domain names, file types etc.
- Comma-separated files, using , . ; : etc.
- Special characters for syntax purposes, \* % & | ? etc. Searching for everything or wild card search

# File Name Canonicalization



`C:\WINDOWS\system32\calc.exe`

or

- `C:\WINDOWS\system32\drivers\..\calc.exe`
- `calc.exe`
- `.\calc.exe`
- `..\calc.exe`
- `\\?\WINDOWS\system32\calc.exe`
- Attacks are called path or directory traversal, using `..` to enter paths not expected by the application, ref Microsoft IIS Unicode vulnerabilities
- Linux allows you to reference a file like `wc -l /etc/passwd` or `wc -l ///etc/////passwd`

# Shell Metacharacters



```
<pre>  
<?php passthru("ping $HOST"); ?>  
</pre>
```

- Misc dangerous shell characters:
- ; separator, execute multiple commands, | pipe, execute multiple commands
- ` ` back ticks, or \$( ) execute a command and insert result
- < > redirect input, output etc.
- Perl: `print `/usr/bin/finger $input{'command'}`;`
- UNIX shell: ``echo hello``
- Microsoft SQL: `exec master..xp_cmdshell 'net user test testpass /ADD'`
- I prefer explicit allow filters (white lists) for filtering metacharacters, if at all possible. Easier for a phone number than name, YMMV

# HTML and XML encoding, plus serialization



- HTML and XML can contain encoded data %20 is a space
- Requests sent over HTTP can contain serialization and de-serialization, basically sending code
- Multiple layers of decoding can result in problems, like double-decode Microsoft IIS vulnerability CVE-2001-0333

# Character sets and unicode



```
GET ../../%c0%af../%c0%afwinnt/system32/cmd.exe?/c+dir
```

- UTF-8 becoming the standard used, example from CVE-2000-0884
- Calls `cmd.exe` with any command from URL
- Example encoding for `/`
- `0x2f`
- `0xC0 0xAF` - the one used above
- `0xE0 0x80 0xAF`
- `0xF0 0x80 0x80 0xAF`

# Django: Unicode data



**Django supports Unicode data everywhere.**

This document tells you what you need to know if you're writing applications that use data or templates that are encoded in something other than ASCII. ... **Useful utility functions**

Because some string operations come up again and again, Django ships with a few useful functions that should make working with string and bytestring objects a bit easier.

Source:

<https://docs.djangoproject.com/en/3.1/ref/unicode/>

# Exercise

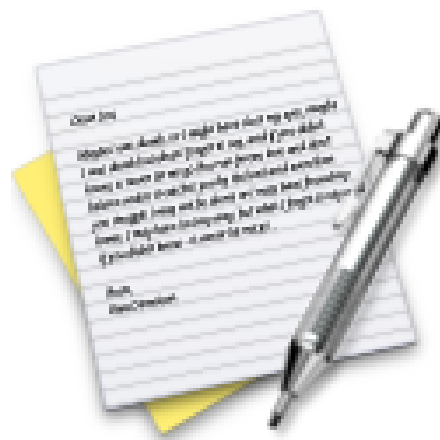


Now lets do the exercise

## Django String Handling 20min

which is number **29** in the exercise PDF.

# Exercise



Now lets do the exercise

## Django ORM 20 min

which is number **30** in the exercise PDF.



# Exercise

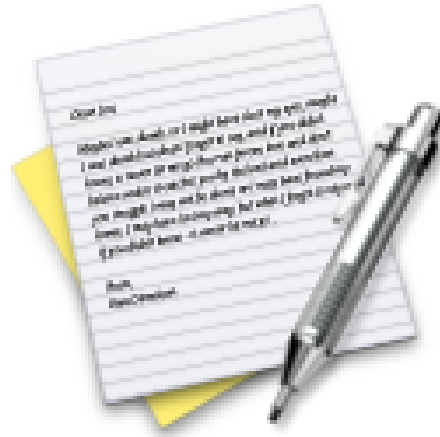


Now lets do the exercise

## Django email validation 30 min

which is number **31** in the exercise PDF.

## Exercise

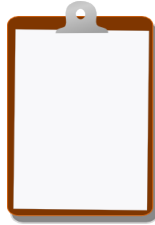


Now lets do the exercise

## Truncate and Encoding Attacks JuiceShop up to 40min

which is number **32** in the exercise PDF.

## For Next Time



Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools