Welcome to

8. Strings and Metacharacters

KEA Kompetence OB2 Software Security 2019

Henrik Lund Kramshøj hlk@zencurity.com @kramse

Slides are available as PDF, kramse@Github

8-strings-and-metacharacters.tex in the repo security-courses

Plan for today

Subjects

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

Exercises

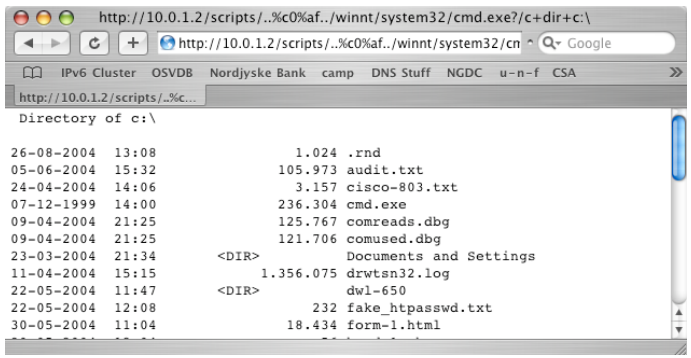
- Recommendations for handling strings, how does Python help, how does Django handle strings, and input validation

Reading Summary

AoSSA chapter 8: Strings and Metacharacters

Also checkout `https://en.wikipedia.org/wiki/C_string_handling` for use when you dont have the book with you.

Goals:



Strings are used in most programs, 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 €€€€

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

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

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 occurence

C String handling

```c
#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
- 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 problemtatic

These were the ones people used in the beginning

30 Years ago in 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.

For more details see doc/doc-txt/cve-2019-15846/ in the source code repository.

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

Better Functions from BSD

- strlcpy, strlcat size-bounded string copying and concatenation
- **strlcpy()** 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

Parsing String Data

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

- Example from the book, if the input is larger than destination pointed to by `tt` then problems can arise
- Character expansion, making output bigger can overflow
- Another `mod_dav` and `mod_mime` vulnerabilities are presented as listings 8-6 and 8-7

Metacharacters

- Null 0x00, special in C, but just another char in higher level languages
- Space
- / used as filename delimiters, and \ in Windows
- . dot used in various ways for domain names, file types etc.
- Comma-seperated 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

Shell Metacharacters

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

- Misc dangerous shell characters, see book for more:
- ; seperator, 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

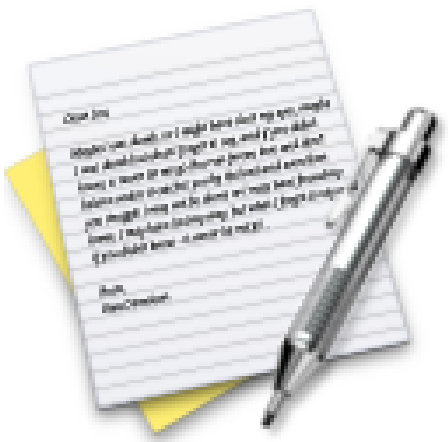
- HTML and XML can contain encoded data %20 is a space
- Requests sent over HTTP can contain serialization and de-serialization, basically sending code
- Multiple layers of decoding can result in problems, like double-decode Microsoft IIS vulnerability CVE-2001-0333

Character sets and unicode

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

- UTF-8 becoming the standard used, book uses the 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

Exercise

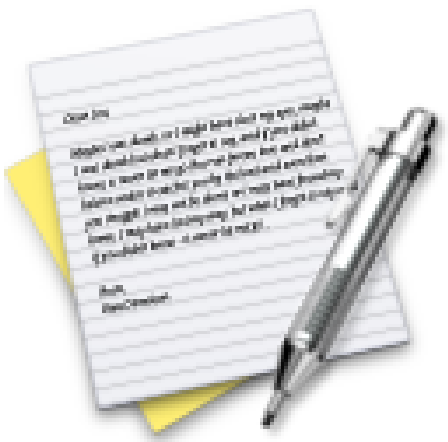


Now lets do the exercise

Truncate and Encoding Attacks JuiceShop up to 40min

which is number **18** in the exercise PDF.

Exercise

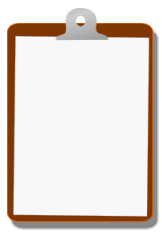


Now lets do the exercise

Django String Handling 20min

which is number **19** 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
Most days have less than 100 pages, but some days may have more!

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools

Buy the books!