Understanding Common Security Exploits

Sam Hartman

Tom Yu

29 January 2004

Today’s Topics

- Countermeasures
- Advanced Techniques

Countermeasures

`noexec_user_stack` (Solaris)

- Kernel-based protection.
- Prevents execution of code on stack.

Weaknesses:

- Not necessary to execute shellcode from stack.
- Can still overwrite stack.
- Non-stack exploits still work.

StackGuard (Linux)

- Inserts “canary” values between stack items.
- Checks canary values in function epilogue.

Weaknesses:

- Early versions didn’t prevent overwriting adjacent locals in same stack frame.
- Can still do targeted overwrites of stack.
- Non-stack exploits still work.

PaX

- Makes all writable pages non-executable.
- Injected code inherently un-runnable; enforced by kernel.
- Recent versions use Address Space Layout Randomization (ASLR).
- Makes returning into libc harder, but not impossible.

Advanced Techniques

Advanced Techniques

- Format Strings
- Return into libc
- Heap Overflows

Format Strings

Format String Vulnerability

```
void f(void)
{
    char str[1024];
    net_read(str, sizeof(str));
    printf(str);
}
```

- Any `printf()` directives in `str` get interpreted.

- More correct is
  `printf("%s", str);`

- Similar functions such as `syslog()` are vulnerable.

Using `printf()` to Read

Inject `printf()` directives to examine stack.

- Use more `printf()` directives than arguments given to `printf()` call.
- `%x` to get frame pointers, return addresses, etc.
- `%s` to get a string whose start address is on the stack somewhere.
- Sometimes, reading is useful: passwords, cryptographic keys, etc.

Writing to Memory

- Little-known directive %n

- Treats its argument as pointer to integer; writes character count there.

- Can choose value to be written, e.g., by using `%.9999d%n`

- Can use %*n*$n (POSIX.1 extension) to choose which argument to overwrite, allowing chaining and more leverage.

- Overwrite return return address, frame pointer, function pointers, etc.

Format String References

- gera, “Advances in format string exploitation,” Phrack 59, 2002.
- Tim Newsham, “Format String Attacks,” Bugtraq, 2000.

Return into libc

Return into libc

- Overwrite return address to point to function in libc.
- Overwrite nearby values in stack to provide “arguments” to libc function that’s being called.
- Can chain multiple libc calls (e.g., `setuid(0)` followed by `exec()`) by fabricating multiple stack frames, including frame pointers.

Return into libc References

- anonymous, "Bypassing PaX ASLR protection," Phrack 59, 2002.
- Nergal, "The advanced return-into-lib(c) exploits," Phrack 58, 2001.

Heap Overflows

The Heap

- Typically, the operating system provides a way for a process to expand its data segment dynamically.

- Implementations of `malloc()` and related functions usually hide this detail from the programmer.

- These functions typically take exclusive control of extending of the data segment

- The managed part of data segment is called the *heap*.

Exploiting the Heap

- Historically, the importance of heap overflow vulnerabilities has been downplayed.

- Even a small heap buffer overflow may lead to arbitrary code execution.

- Function pointers stored in the heap can be overwritten.

- Many `malloc()` implementations share features that make them vulnerable to exploitation in case of overflows.

Heaps and Function Pointers

- Function pointers can be stored in data structures on the heap or stack.
- A buffer overflow can replace one of these function pointers.
- The hard problem is knowing where the function pointer should point; it can point to the stack, to a buffer on the heap, or to a function in existing code.

Function Pointer Example

PAM uses a callback structure similar to the following to track module data.

```
struct pam_data {
    char name[32];
    void *data;
    void (*cleanup)(pam_handle_t *pamh, void *data,
                    int error_status);
    struct pam_data *next;
};
```

Common Features of `malloc()` Implementations

- In-band storage of management information.
- Overwriting this in-band management information can lead to misbehavior of the `malloc()` implementation.
- Typically, the management information for a chunk immediately precedes the address returned to the caller.

GNU libc's `malloc()` implementation (dlmalloc)

- Implementation by Doug Lea (hence "dl").
- dlmalloc uses a "boundary tag" method of managing allocated chunks.

The boundary tag is declared like:

```
struct malloc_chunk {
  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      prev_size;
  /* Size in bytes, including overhead. */
  INTERNAL_SIZE_T      size;

  /* double links -- used only if free. */
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
};
```

Quirks of Boundary Tag Use

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;
  INTERNAL_SIZE_T      size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
};
```

- In an allocated chunk, user data begins at `fd` and also overwrites `bk`.
- If the previous chunk is allocated, its user data is allowed overwrite `prev_size` of the current chunk.
- The least significant bit of `size` is set if the previous chunk is in use. This is possible due to alignment requirements on `size`.
- `fd` and `bk` link freed chunks into doubly-linked circular lists.

Exploiting dlmalloc

- Overflowing a buffer allocated by `malloc()` will overwrite the boundary tag of the following chunk.

- Overwriting the `fd` or `bk` pointers of a freed chunk can cause `malloc()` to write to arbitrary memory when it attempts to allocate that chunk.

- Under certain circumstances, a single byte overflow may be sufficient to allow for exploitation.

Single-Byte Overflows

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size; /* O */
  INTERNAL_SIZE_T      size;
  struct malloc_chunk* fd; /* O */
  struct malloc_chunk* bk; /* O */
};
/* O -> user data allowed to overwrite */
```

- On little-endian architectures, overflowing a chunk by one byte overwrites least significant byte of `size` for following chunk.
- Least significant bit of a chunk's `size` is set if the previous chunk is in use.
- Writing a byte that clears the "in-use" bit causes `malloc()` implementation to treat the previous chunk as free.

Single-Byte Overflows (cont'd)

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size; /* O */
  INTERNAL_SIZE_T      size;
  struct malloc_chunk* fd; /* O */
  struct malloc_chunk* bk; /* O */
};
/* O -> user data allowed to overwrite */
```

- dlmalloc consolidates freed chunks to avoid fragmentation.
- Last four bytes of overflowed chunk's data overlap `prev_size` of following chunk; `prev_size` determines location of "freed" chunk's boundary tag.
- Fabricate bogus boundary tag data for overflowed chunk, including bogus `fd` and `bk` pointers.

Heap Exploitation References

- anonymous, “Once upon a `free()`,” Phrack 57, 2001.
- jp, “Advanced Doug lea’s malloc exploits,” Phrack 61, 2003.
- Michel “MaXX” Kaempf, “Vudo malloc tricks,” Phrack 57, 2001.

Additional Resources

Bugtraq: `http://www.securityfocus.com/`

Phrack: `http://www.phrack.org/`

grsecurity (PaX): `http://grsecurity.net/`

These slides are available at
`http://www.mit.edu/iap/exploits/exploits02.pdf`
Course Home Page:
`http://www.mit.edu/iap/exploits/`