Buffer Overruns Explained

Shachar Shemesh

Security Consultant

http://www.shemesh.biz/

What are They?

- Any time an attacker can write more data than the buffer can hold.
- Two major types:
  - Stack overrun
  - Heap overrun

Stack Overruns

- The oldest trick in the book.
- Exploitation is almost a game of trivially applying a well known technique.
- The single most exploited vulnerability.
  - The first worm, called the “Morris Worm”, used a stack overrun in “Sendmail” – 1988.

Heap Overruns

- Considered dangerous for ages.
  - One would have to “get lucky” with a convenient pointer.
- Only mid 2002 – cookie-cut exploitation method.
- Related cousin – double free errors.

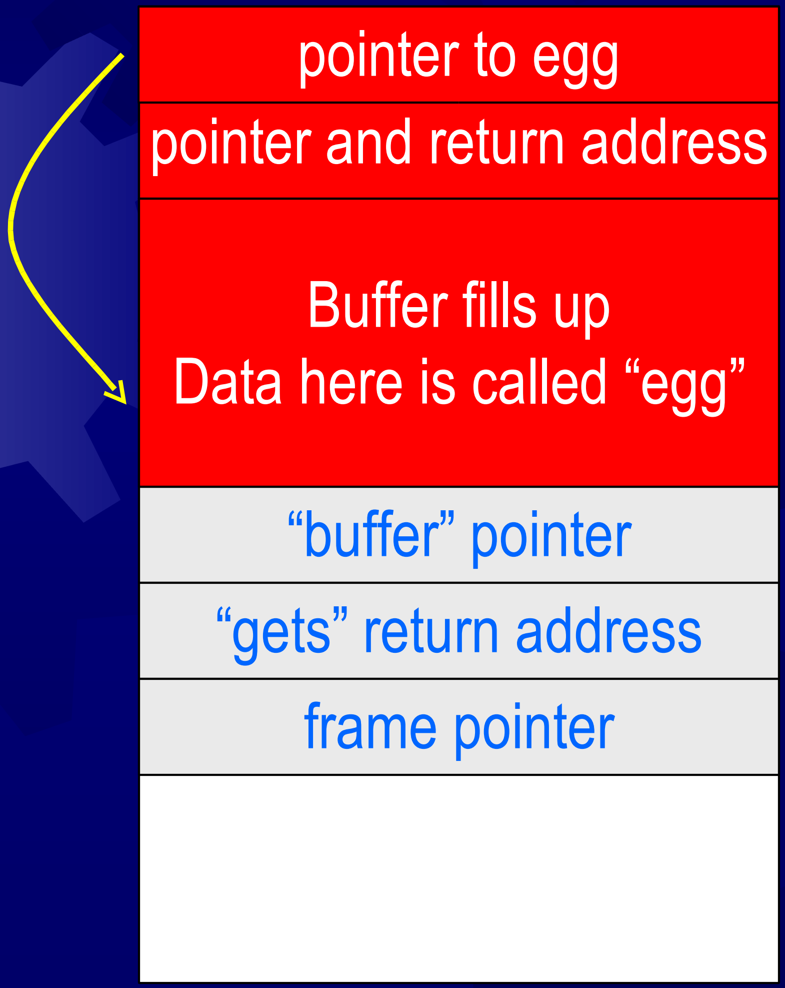
Stack Overruns – How it Works

A few things to understand:

- The stack usually grows downwards.
- The stack frame in “C” – arguments, return address, base pointer, automatic vars.
- Non of this practically matters – exploitation is usually possible even if the above is wrong.

Stack Overrun – Arbitrary Code Execution HOWTO

The Stack



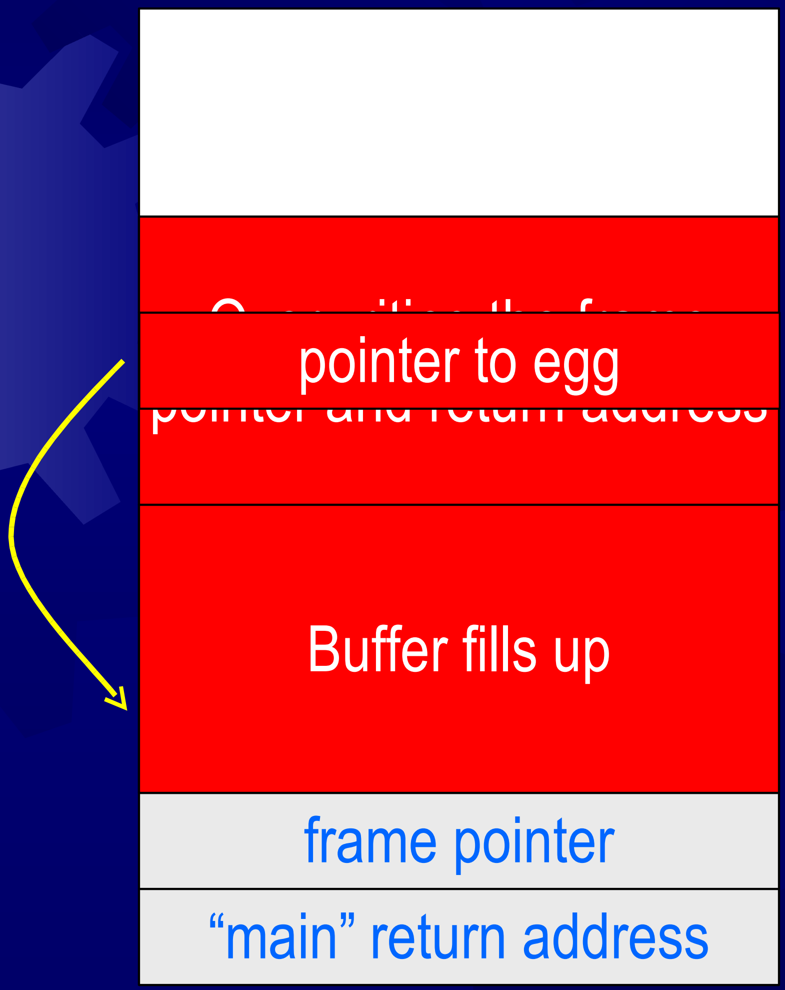

```
main()
{
    char buffer[250];

    gets(buffer);
    printf(buffer);
    printf(“\n”);
}
```

Analysis

- When "main" tries to return, the execution will flow into the buffer.
- The egg has to be relocateable code.
- The egg has to avoid certain characters.
  - In "gets" case – newline.
  - Avoiding any single character is no problem.
  - There is work (nearly complete) on printable only egg for i386.

Upward Growing Stack

The Stack




```
main()
{
    char buffer[250];

    gets(buffer);
    printf(buffer);
    printf("\n");
}
```

Heap Overruns – Until 2002

- Analyze the heap – search for convenient pointers.
- Exploit code highly dependant on exact program state.
- Even so – extremely dangerous to assume any given buffer overrun is safe.

Heap Overruns – 2002 Edition

- The head is allocated in one contiguous block.
- Management of the individual allocation blocks is done with a data structure.
  - Usually a balanced or a 2/3 tree.
  - The pointers for that data structure are maintained in the same area as the heap.
- Writing past the end of a buffer change this structure.

Heap Overruns – cont.

- When an application frees memory free heap sections are merged.
- As a result, an attacker can cause arbitrary values to be written to arbitrary locations!
- The road from here to arbitrary code execution is not long (demo next week).

Known Dangerous Functions

- *sprintf*
  - Field length specifiers can prevent the problem.
  - Use the alternative *snprintf*.
- Occasionally – *scanf* and *fscanf*
  - Again – limit each field's length.
- The str* functions – *strcat*, *strcpy*
  - Use *strncat* and *strncpy* instead.
    - Watch out for the usage!
- *gets*
- Your own loops.

Examples of Dangerous Usage: *scanf* and *fscanf*

```
int main( int argc, char *argv[] )
{
      char buffer[250];

      scanf("%s", buffer );
      printf( "%s\n", buffer );

      return 0;
}
```

scanf and *fscanf* vulnerabilities (cont.)

- There is no difference, in principle, between the previous example, and the one using gets.
- The egg needs to avoid the space and newline characters, but writing such eggs is an everyday practice for an experienced cracker.
- Changing the scanf line to read '`scanf ("%250s", buffer);`' would have solved the problem.

sprintf vulnerabilities

- Assuming that the following is a set-UID program:

```
int main( int argc, char *argv[] )
{
        char buffer[250];


        sprintf(buffer, "Usage: %s <name>\n", argv[0]);
        printf( buffer );
...
}
```

sprintf vulnerabilities

- In the previous example, argv[0] is used to quote the program's name.
  - argv[0] is actually supplied as a parameter to the kernel function "*execve*". There is no limit to it's length.
- *sprintf* buffer-overrun vulnerabilities usually stem from two sources:
  - Formatting user supplied arguments, or environment variables (registry).
  - incorrect calculation of total buffer length when combining buffers.

str* functions

```
int main( int argc, char *argv[] )
{
	char buffer[250];

	strcpy(buffer, argv[1]);
	printf( “%s\n”, buffer );

	return 0;
}
```

str* functions (cont.)

- No need to explain why this is dangerous.
- Most str* functions have a corresponding strn* functions (i.e. – *strncpy* instead of *strcpy*).
- Notice, however, that the strn* functions have very confusing interface!!

The “*gets*” Function

```
int main( int argc, char *argv[] )
{
      char buffer[250];


      gets(buffer);
      printf( “%s\n”, buffer );


      return 0;
}
```

The “*gets*” Function (cont.)

- Always gets its data from an external source (stdin), which is rarely secure.
- Has no facility to check the buffer’s length.
- Is so dangerous, many modern linkers issue a warning if it is referenced.
  - On *BSD systems – runtime warning.
- Use “`fgets( buffer, buff_size, stdin);`” for identical results with boundaries checking.

Your Own Loops

What's wrong with this program?

```
int main( int argc, char *argv[] )
{
        char buffer[250];
        int i,c;
        for( i=0; (c=getchar())!=EOF && c!='\n' && i<250; ++i )
                buffer[i]=c;

        buffer[i]='\0';
        printf("%s\n", buffer);

        return 0;
}
```

Your Own Loops (cont.)

- If the input length is 250 characters or more, a single byte after the end of the buffer is overwritten with NULL.
- With an upward growing stack, and a little endian machine (such as Intel), this means overwriting the LSB of the pointer right after the buffer with zero.
- With the buffer size occupying most (but not all) of the previous 256 block, there is a very high probability that the new pointer points back into the buffer.
- There is a good chance that this bug is exploitable!

Cast screwups

```
void func(char *dnslabel)
{
  char    buffer[256];
  char    *indx = dnslabel;
  int     count;

  count = *indx;
  buffer[0] = '\x00';

  while (count != 0 && (count + strlen (buffer)) < sizeof (buffer) - 1)
  {
    strncat (buffer, indx, count);
    indx += count;
    count = *indx;
  }
}
```

First byte at *dnslabel is 0x80 = -128

Gets expanded to 0xFFFFF80

signed comparison passes

almost arbitrary length string is appended

Further Reading

The extra material is for anyone who is interested in deeper understanding of exploiting buffer overruns

- Smashing the stack for fun and profit – http://www.phrack.org/show.php?p=49&a=14
- Exploiting heap overruns – http://www.phrack.org/show.php?p=57&a=9

Next Meeting (in two weeks)

- Explanation of format strings exploitation methods.
- Live demonstration of “from scratch” development of a simple exploit code.
  - Stack overrun.
  - Format string.

Available Online

- This presentation (as well as others soon to follow) is available in an all-browser digestible form at http://www.shemesh.biz/lectures

Questions Time