Writing Optimized and Secure Code

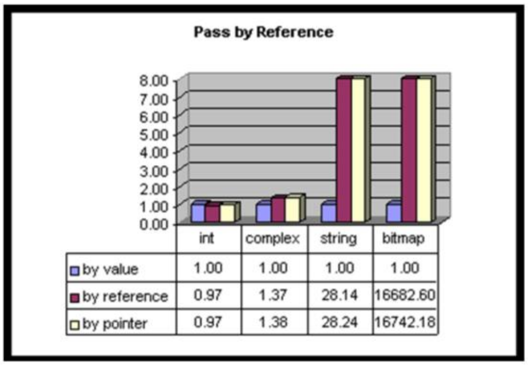
ITP 435 – Spring 2016
Week 5, Lecture 2
Lecturer: Sanjay Madhav

Optimizing Code "As You Go"

- If you follow some simple rules, you can avoid some common pitfalls.
- These rules will help you write code that's more efficient!

Good Rules to Follow

- Non-basic types should be passed by reference when possible:



	int	complex	string	bitmap
by value	1.00	1.00	1.00	1.00
by reference	0.97	1.37	28.14	16682.60
by pointer	0.97	1.38	28.24	16742.18

From: http://www.tantalon.com/pete/cppopt/main.htm

Good Rules to Follow (cont'd)

- Postpone variable declarations when possible:

```
// Declare Outside (b is true half the time)
T x;
if (b) {
    x = t;
    // Do stuff...
}

// Declare Inside (b is true half the time)
if (b) {
    T x = t;
    // Do stuff...
}
```

Good Rules to Follow (cont'd)

- Prefer operator+= to operator+ (and like operators)

```
struct Vector2
{
    float X;
    float Y;

    Vector2& operator+=(const Vector2& rhs)
    {
        X += rhs.X;
        Y += rhs.Y;

        return *this;
    }

    Vector2 operator+(const Vector2& rhs)
    {
        Vector2 temp(*this);
        temp += rhs;
        return temp;
    }
};
```

USC Viterbi School of Engineering

University of Southern California

Operator+ has to return a copy by value

Good Rules to Follow (cont'd)

- Use prefix instead of postfix for non-basic types:

```
const T T::operator++ (int) // postfix
{
    T orig(*this); // Have to make copy in postfix
    ++(*this); // call prefix operator
    return (orig);
}
```

Good Rules to Follow (cont'd)

- Prefer explicit constructors

```
class pair
{
    double x, y;
public:
    pair(const string& s) { . . . }
    bool operator == (const pair& c) const { . . . }
};

// Without explicit can do:
pair p;
string s;
if (p == s) { . . . }
```

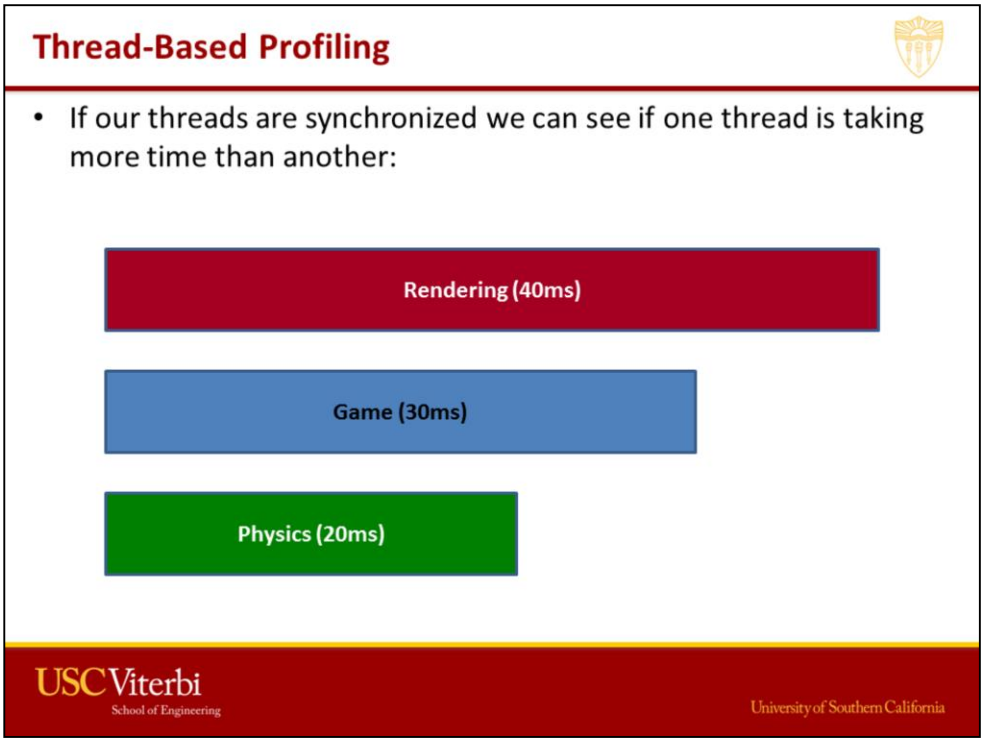
Good Rules to Follow (cont'd)

- If we change the constructor to explicit

```
explicit pair(const string& s) { . . . }

// Won't work!
pair p;
string s;
if (p == s) { . . . }
```

80/20 Rule

- "80-20" rule: 80% of your execution time is spent in 20% of your code



In this case, we clearly want to focus on optimizing our rendering thread first, as it’s slowing everything else down.

Profiler

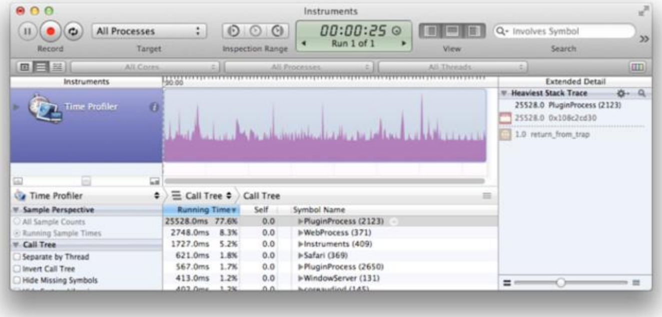
- Profiling programs can allow us to see which functions take up the most time…
- From Intel VTune:

/Function /Call Stack	CPU Time
⊞ initialize_2D_buffer	11.768s
⊟ grid_intersect	5.916s
⊞ ↖ intersect_objects	5.431s
⊞ ↖ grid_intersect ← intersect_objects	0.485s
⊞ sphere_intersect	5.044s

Line	Source	CPU Time
579	`cur = g->cells[voxindex];`	0.204s
580	`while (cur != NULL) {`	0.048s
581	`if (ry->mbox[cur->obj->id] != r`	1.611s
582	`ry->mbox[cur->obj->id] = ry->`	1.025s
583	`cur->obj->methods->intersect(`	1.098s

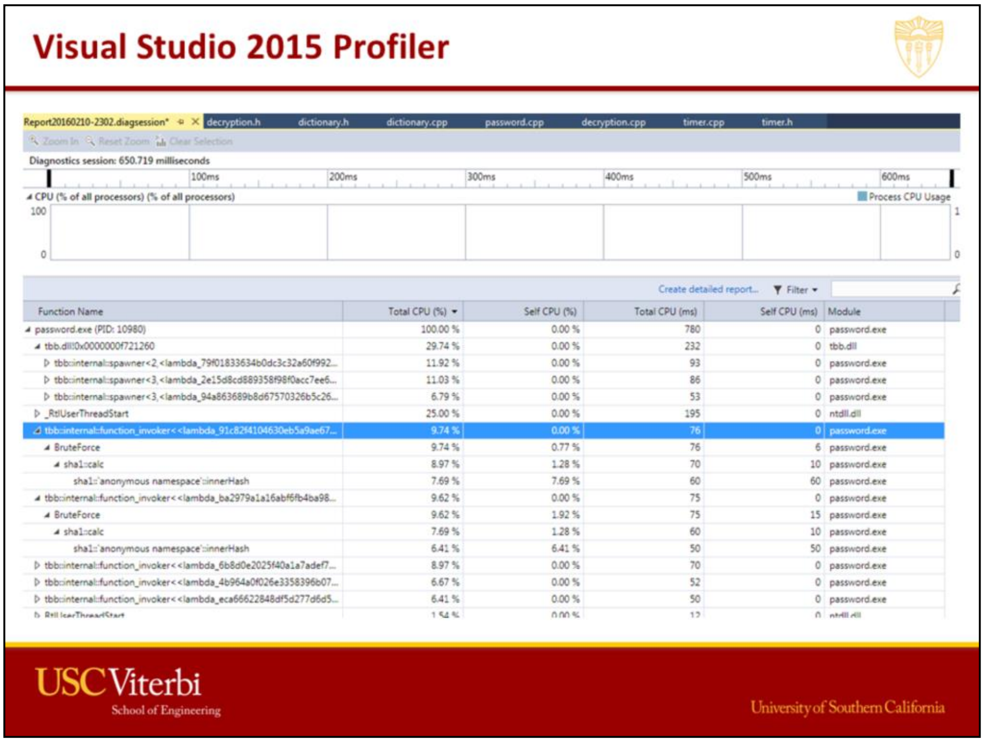
Xcode Profiler

- Product>Profile, select "Time Profiler"



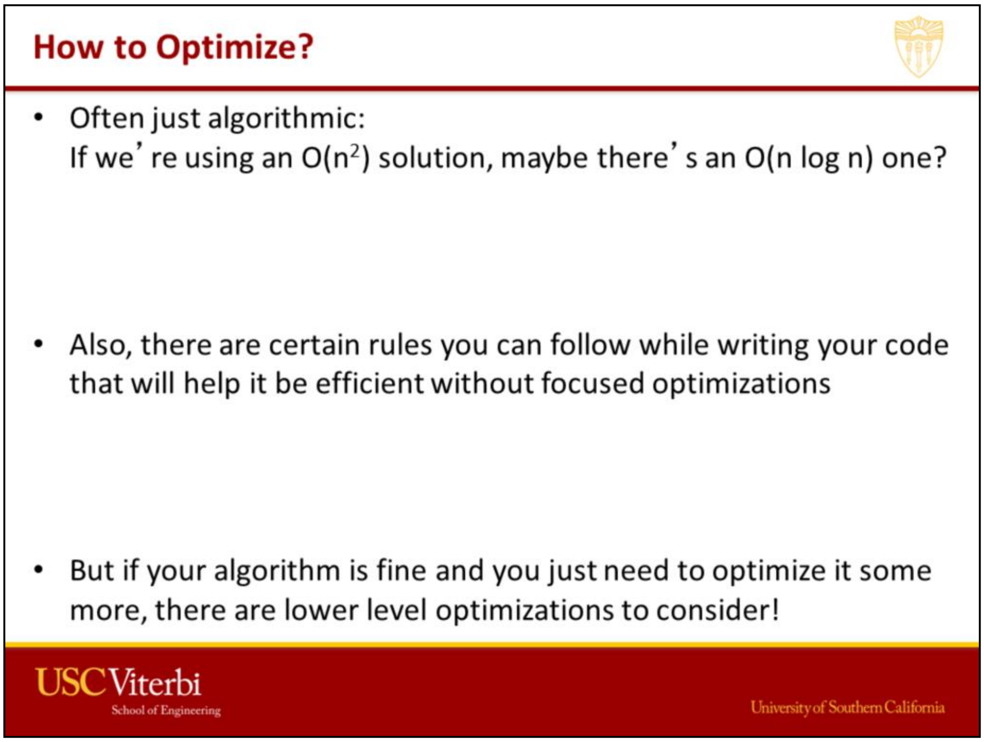
USC Viterbi
School of Engineering

University of Southern California

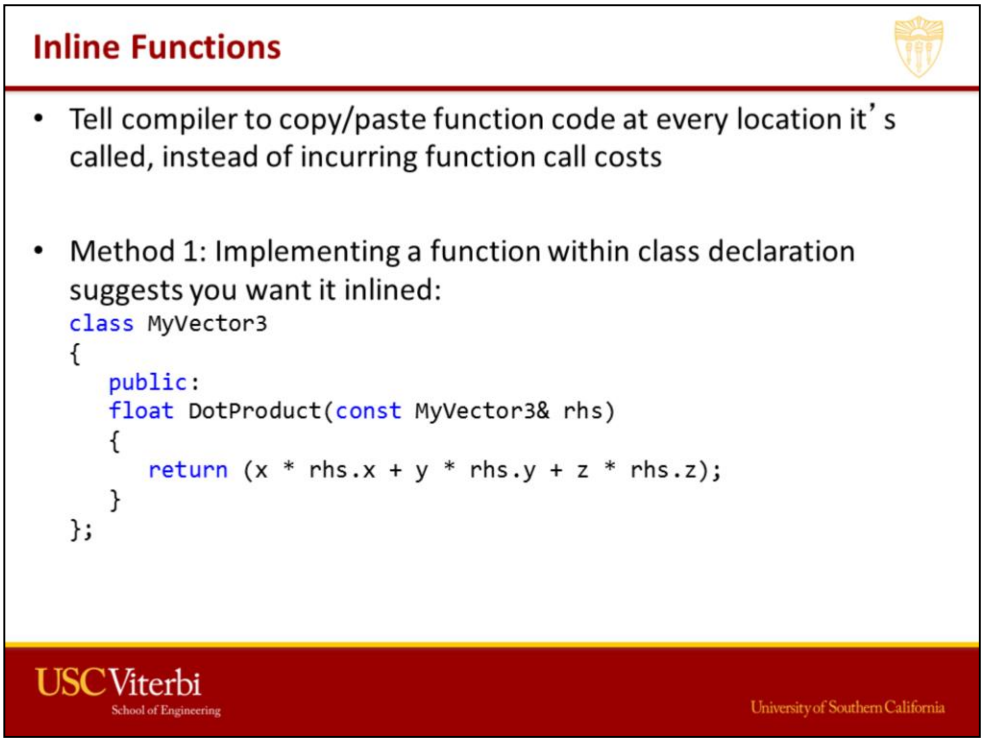


Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)	Module
password.exe (PID: 10980)	100.00 %	0.00 %	780	0	password.exe
tbb.dll!0x0000000f721260	29.74 %	0.00 %	232	0	tbb.dll
tbb::internal::spawner<2,<lambda_79f01833634b0dc3c32a60f992...	11.92 %	0.00 %	93	0	password.exe
tbb::internal::spawner<3,<lambda_2e15d8cd889358f98f0acc7ee6...	11.03 %	0.00 %	86	0	password.exe
tbb::internal::spawner<3,<lambda_94a863689b8d67570326b5c26...	6.79 %	0.00 %	53	0	password.exe
_RtlUserThreadStart	25.00 %	0.00 %	195	0	ntdll.dll
tbb::internal::function_invoker<<lambda_91c82f4104630eb5a9ae67...	9.74 %	0.00 %	76	0	password.exe
BruteForce	9.74 %	0.77 %	76	6	password.exe
sha1::calc	8.97 %	1.28 %	70	10	password.exe
sha1::`anonymous namespace'::innerHash	7.69 %	7.69 %	60	60	password.exe
tbb::internal::function_invoker<<lambda_ba2979a1a16abf6fb4ba98...	9.62 %	0.00 %	75	0	password.exe
BruteForce	9.62 %	1.92 %	75	15	password.exe
sha1::calc	7.69 %	1.28 %	60	10	password.exe
sha1::`anonymous namespace'::innerHash	6.41 %	6.41 %	50	50	password.exe
tbb::internal::function_invoker<<lambda_6b8d0e2025f40a1a7adef7...	8.97 %	0.00 %	70	0	password.exe
tbb::internal::function_invoker<<lambda_4b964a0f026e3358396b07...	6.67 %	0.00 %	52	0	password.exe
tbb::internal::function_invoker<<lambda_eca66622848df5d277d6d5...	6.41 %	0.00 %	50	0	password.exe
RtlUserThreadStart	1.54 %	0.00 %	12	0	ntdll.dll

Debug>Start Diagnostic Tools Without Debugging…



We've already talked about some of our function inlining techniques



You may want to inline a function if it's small and is being called a lot. In this case, there can be a significant performance gain by not calling the function.

Note that virtual functions called virtually *can never be inlined*. This is another hidden cost of virtual functions.

These constructs only *suggest* inlining. The compiler will still ultimately decide whether or not it wants to do it.

Inline Functions, cont'd

- Method 2: For standalone functions, use the "inline" for a suggestion:

```
inline void MyInlineFunction()
{
   // Do something
}
```

Force Inlining

- Visual Studio has a way to “force” inlining, which will inline in almost all cases:

```
__forceinline void MyInlineFunction()
{
   // Do something
}
```

- However, there are still some cases where it won’t be inlined:
  - Debug build
  - Recursive function
  - Virtual function called virtually
  - Function pointer call
  - And a couple of other really rare cases

Loop Unrolling

- Regular loop:

```
// This calls an arbitrary function 100 times
for (int x = 0; x < 100; x++)
{
    func();
}
```

USC Viterbi School of Engineering

University of Southern California

If this needs to be high-performance code, the loop as written spends quite a few instructions just incrementing and checking the loop. Also does a lot of branching.

Loop Unrolling, cont'd

- This is a partially "unrolled" version of the same loop:

```
// This calls an arbitrary function 100 times
for (int x = 0; x < 100; x+=5)
{
    func();
    func();
    func();
    func();
    func();
}
```


Disadvantage: Increases size of code

Return Value Optimization

- When returning a class by value, on some compilers it may be more efficient to return an *unnamed* class as opposed to a named one

```
template <class T> T Original(const T& tValue)
{
    T tResult; // named object; optimization potential low
    tResult = tValue;
    return (tResult);
}

template <class T> T Optimized(const T& tValue)
{
    return (T(tValue)); // unnamed; optimization potential high
}
```

USC Viterbi School of Engineering

University of Southern California

Named Return Value Optimization

Many newer versions of compilers support it. From Visual Studio documentation:

"NRVO eliminates the copy constructor and destructor of a stack-based return value. This optimizes out the redundant copy constructor and destructor calls and thus improves overall performance. It is to be noted that this could lead to different behavior between optimized and non-optimized programs."

Named Return Value Optimization (cont’d)

```
#include <stdio.h>
struct RVO
{
     RVO(){printf("I am in constructor\n");}
     RVO (const RVO& c_RVO) {printf ("I am in copy constructor\n");}
     ~RVO(){printf ("I am in destructor\n");}
     int mem_var;
};
RVO MyMethod (int i)
{
     RVO rvo;
     rvo.mem_var = i;
     return (rvo);
}
int main()
{
     RVO rvo;
     rvo=MyMethod(5);
}
```

NRVO Output

- Without NRVO (cl /Od sample1.cpp), the expected output would be:

```
I am in constructor
I am in constructor
I am in copy constructor
I am in destructor
I am in destructor
I am in destructor
```

- With NRVO (cl /O2 sample1.cpp), the expected output would be:

```
I am in constructor
I am in constructor
I am in destructor
I am in destructor
```

This code works but…

```
#include <iostream>
int main() {
   bool allowAccess = false;
   char buffer[16];
   std::cout << "Enter Password:";
   std::cin >> buffer;

   if (std::strcmp(buffer, "trustno1") == 0) {
      allowAccess = true;
   } else {
      std::cout << "Bad password." << std::endl;
   }

   if (allowAccess) {
      std::cout << "Access granted." << std::endl;
      // DO STUFF
   }

   return 0;
}
```

USC Viterbi School of Engineering

University of Southern California

Time for a demonstration…

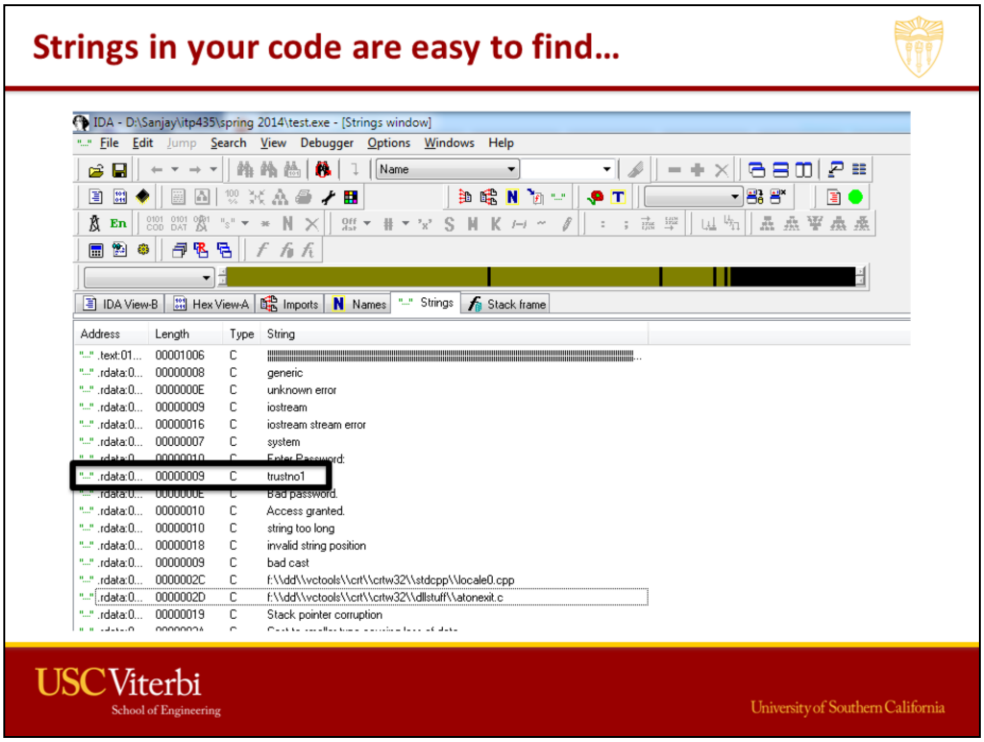
Some sample input to the program…

```
Administrator: C:\Windows\system32\cmd.exe

D:\Sanjay\itp435\spring 2014>test
Enter Password:trustno1
Access granted.

D:\Sanjay\itp435\spring 2014>test
Enter Password:aga;kgja
Bad password.

D:\Sanjay\itp435\spring 2014>test
Enter Password:11111111111111111111
Bad password.
Access granted.

D:\Sanjay\itp435\spring 2014>
```

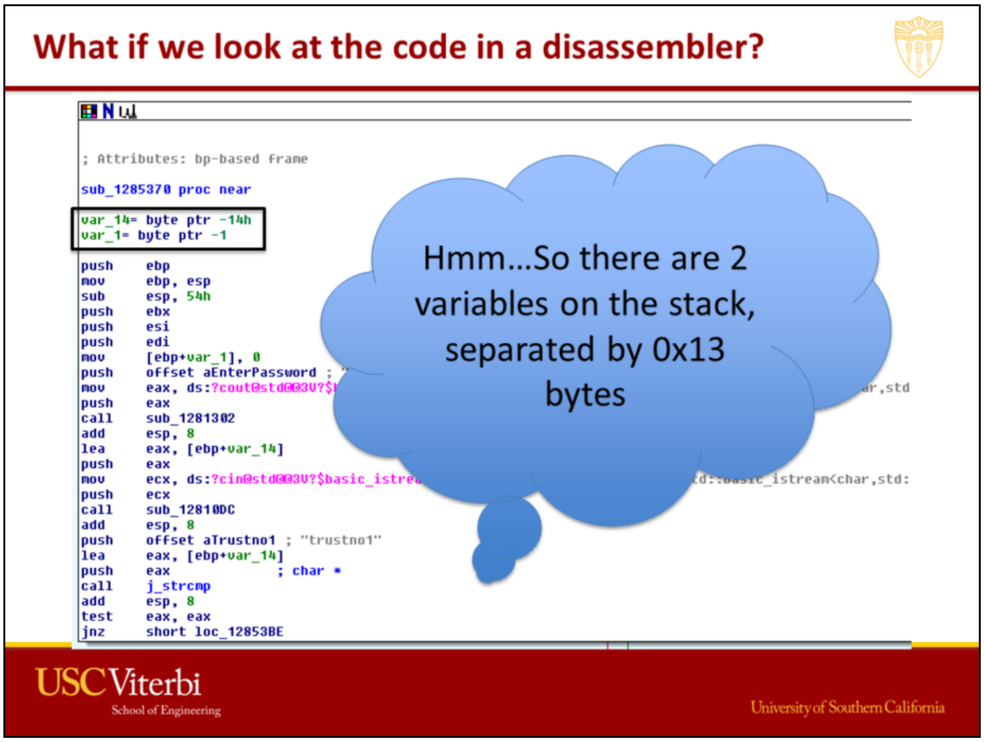
USC Viterbi School of Engineering

University of Southern California

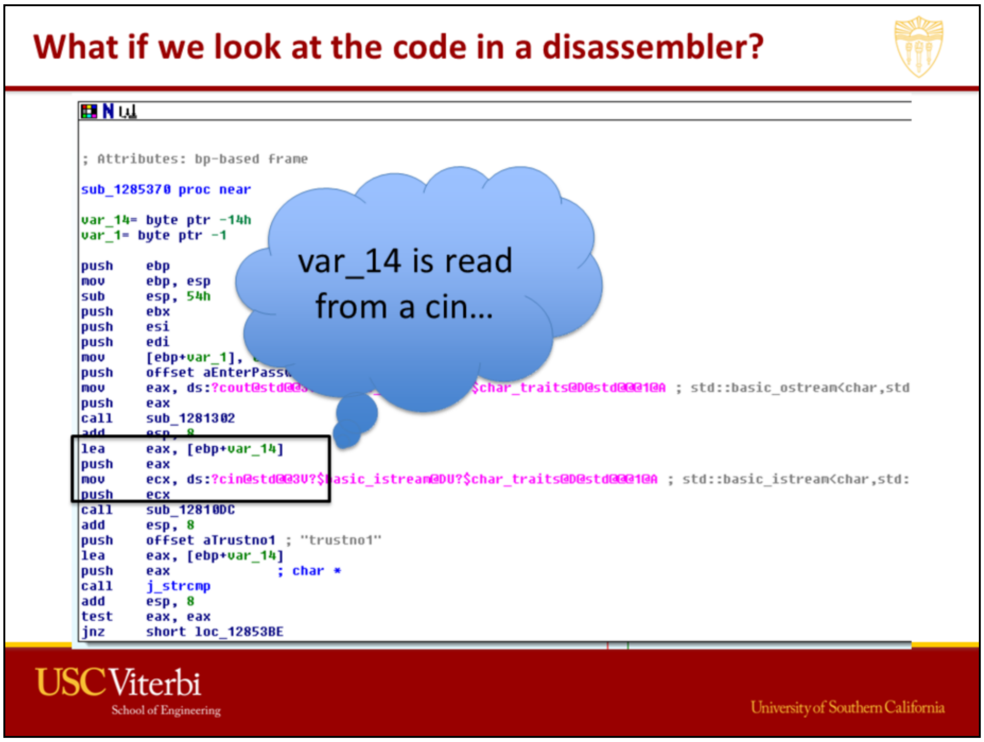
Why? Well, first let me show you something…



Lesson: Unencrypted strings are trivial to locate in a disassembler

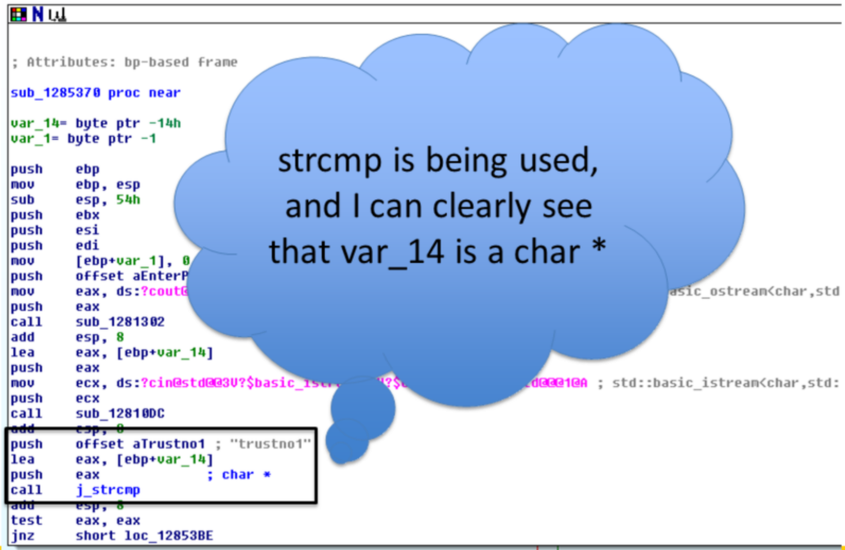


I don't expect you to know the assembly, of course



I don’t expect you to know the assembly, of course

What if we look at the code in a disassembler?



What if we look at the code in a disassembler?

Stack Variable Allocation

- In our program's case, the stack variables are something like this:

Address	
0x00	`char buffer[16]`
0x04	
0x08	
0x0C	
0x10	???padding???
0x14	`bool allowAccess`

USC Viterbi School of Engineering

University of Southern California

Normal usage case…

- If we write 15 or less characters to buffer, we're okay:

Address				
0x00	't'	'r'	'u'	's'
0x04	't'	'n'	'o'	'1'
0x08	'\0'			
0x0C				
0x10	???padding???			
0x14	1	0	0	0

"Malformed string" Usage Case

- If we write exactly 21 letters we'll get something like:

Address				
0x00	'1'	'1'	'1'	'1'
0x04	'1'	'1'	'1'	'1'
0x08	'1'	'1'	'1'	'1'
0x0C	'1'	'1'	'1'	'1'
0x10	'1'	'1'	'1'	'1'
0x14	'1'	'\0'	0	0

The least significant byte of allowAccess is now non-zero, which is true!!!

How to fix this problem?

```
#include <iostream>
#include <string>
int main() {
   bool allowAccess = false;
   std::string buffer;
   std::cout << "Enter Password:";
   std::cin >> buffer;

   if (buffer == "trustno1") {
      allowAccess = true;
   } else {
      std::cout << "Bad password." << std::endl;
   }

   if (allowAccess) {
      std::cout << "Access granted." << std::endl;
      // DO STUFF
   }
   return 0;
}
```

USC Viterbi School of Engineering University of Southern California

The easy answer is use std::string instead, as it cannot be susceptible to stack-based buffer overruns, since the storage is on the heap and it dynamically will resize if too many characters in the input

If you really must use c-style strings…

- In Visual Studio, there are so-called "Secure" CRT functions

http://msdn.microsoft.com/en-us/library/8ef0s5kh(v=vs.90).aspx

- For example, we could change our previous example to be:

```
void foo (char *bar)
{
     char  c[12];
     strcpy_s(c, bar);
}
```

- If it looks the same, the secret is in templates:

```
template <size_t size>
errno_t strcpy_s(
   char (&strDestination)[size],
   const char *strSource );
```

USC Viterbi School of Engineering

University of Southern California

http://msdn.microsoft.com/en-us/library/8ef0s5kh(v=vs.90).aspx

And buffer overruns can be a lot worse

- A more complex, but possible buffer overrun exploit is to execute arbitrary code injected by the attacker

- ***shellcode*** can be used in a string to invoke a shell session with (ideally) admin/root access:

```
char shellcode[ ] = "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

Double Free Vulnerability

- Freeing/deleting a pointer twice corrupts the underlying heap and can potentially allow an attacker to cause a buffer overflow

```
char* ptr = new char[SIZE];
...
if (abrt)
{
    delete[] ptr;
}
...
delete[] ptr;
```

Double Free Vulnerability (cont’d)

- Solution: Set pointer to nullptr when deleted. It is safe to delete a nullptr.

```
char* ptr = new char[SIZE];
...
if (abrt)
{
      delete[] ptr;
      ptr = nullptr; // VULNERABILITY FIXED
}
...
delete[] ptr;
```

https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/

http://arstechnica.com/security/2015/12/researchers-confirm-backdoor-password-in-juniper-firewall-code/

https://code.google.com/p/google-security-research/issues/detail?id=675

http://arstechnica.com/security/2015/08/how-security-flaws-work-the-buffer-overflow/

http://thehackernews.com/2014/04/heartbleed-bug-explained-10-most.html