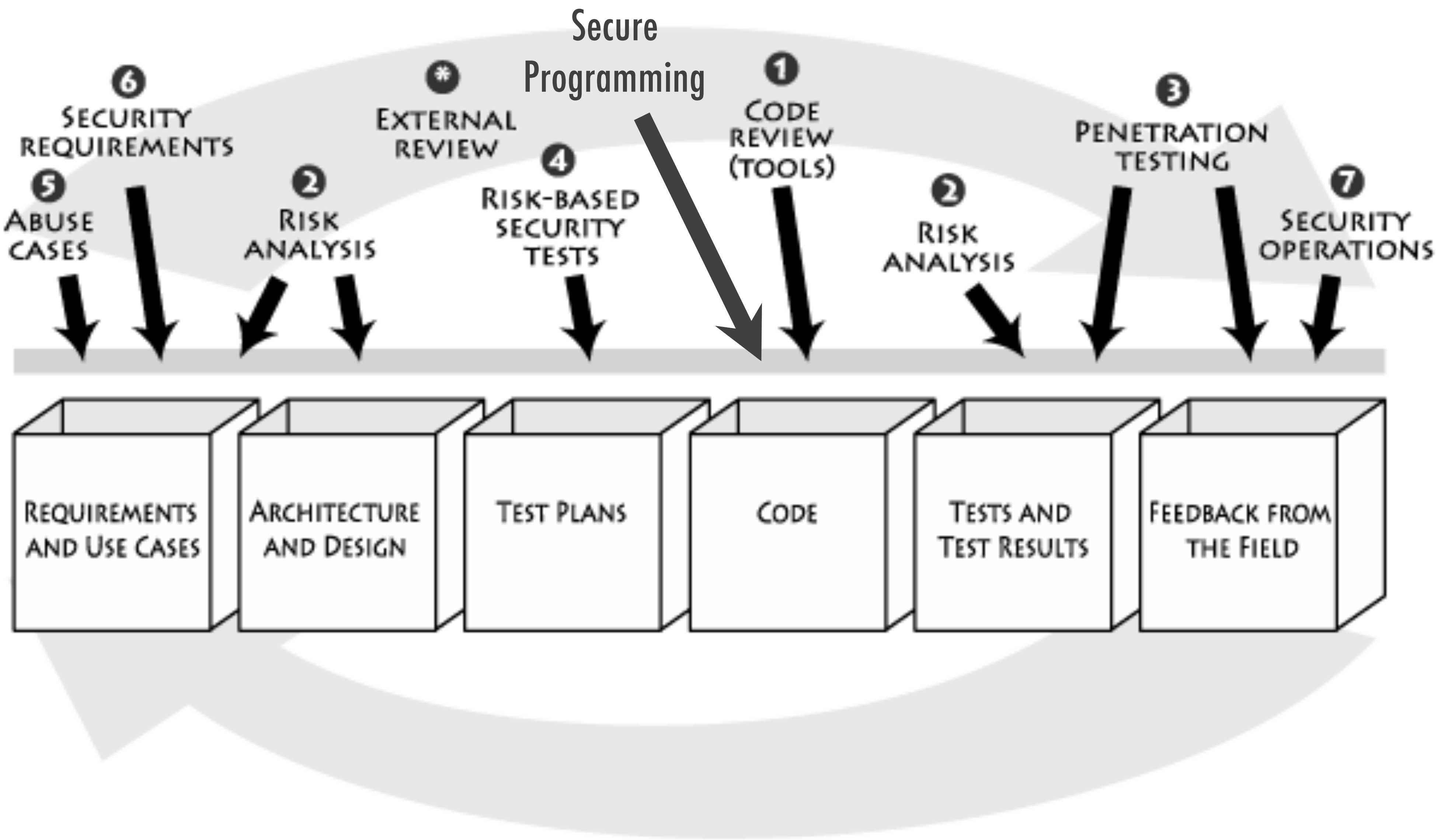
Secure Software Development Life Cycle

Secure SDLC


Secure Programming
6 Security Requirements
5 Abuse Cases
* External Review
2 Risk Analysis
4 Risk-Based Security Tests
1 Code Review (Tools)
2 Risk Analysis
3 Penetration Testing
7 Security Operations
Requirements and Use Cases
Architecture and Design
Test Plans
Code
Tests and Test Results
Feedback from the Field

Security Requirements Engineering: Two Approaches

- Security By Certification
  - Static & Dynamic Analysis
  - Information Flow Control
  - Best Practices, Security Guidelines

Fuzzer / Code Scanners

Static Analysis / Code Audit

Security testing / Pentesting

Common Criteria / EAL

Secure Programming Guidelines

- Security By Design
  - Security Objectives
  - Threat Analysis

Security Architectures

Security Properties

Access Control / Cryptographic Protocols

Security Requirements Engineering: Two Approaches

- Security By Certification
  - Static & Dynamic Analysis
  - Information Flow Control
  - Best Practices, Security Guidelines
- Security By Design
  - Security Objectives
  - Threat Analysis

Defensive Programming

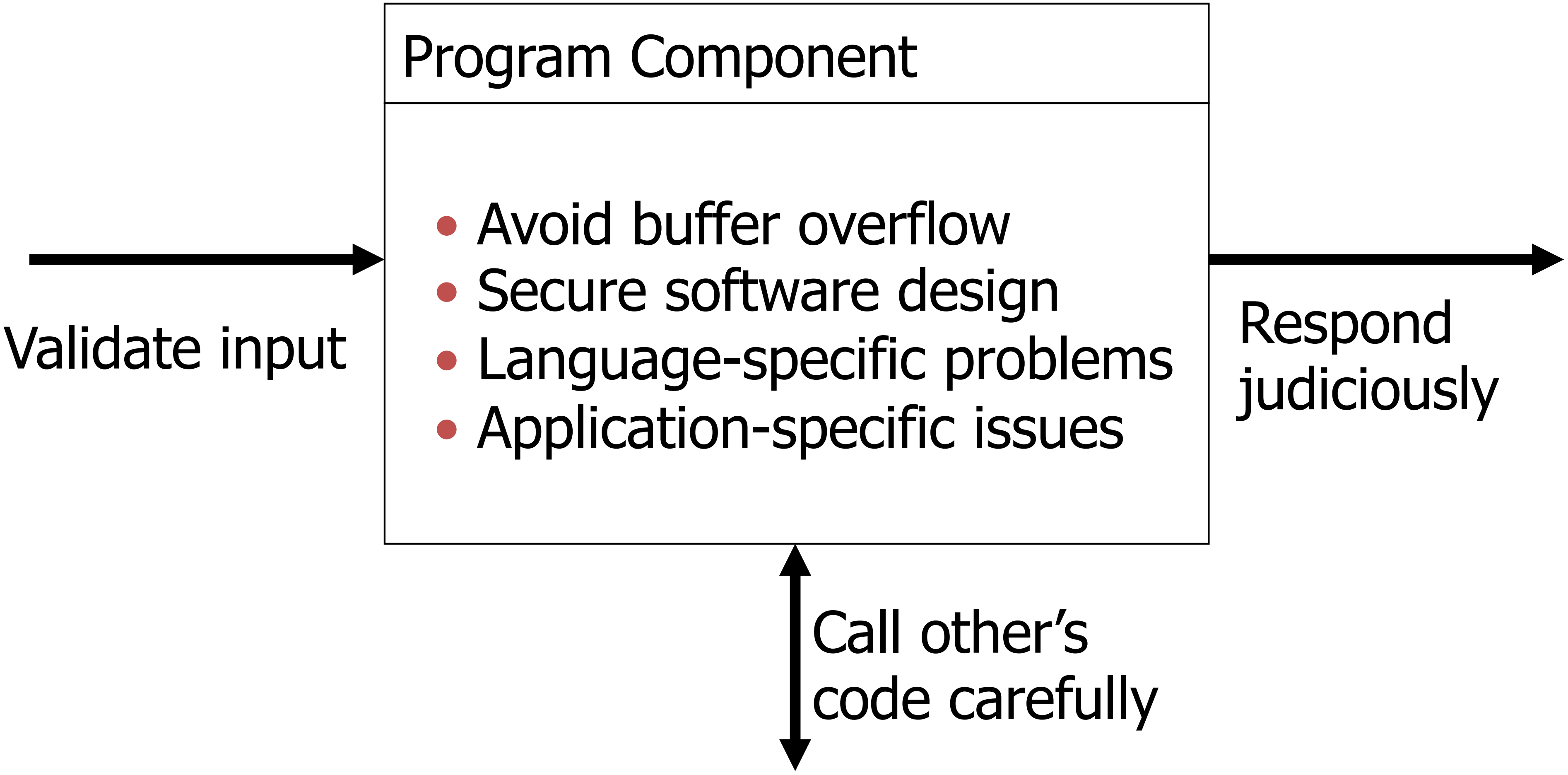
Principles [Viega and McGraw]

- Secure the weakest link
- Practice defense in depth
- Fail securely
- Follow the principle of least privilege
- Compartmentalize
- Keep it simple
- Promote privacy
- Remember that hiding is hard
- Be reluctant to trust
- Use your community resources

Secure the weakest link

- Think about possible attacks
  - How would someone try to attack this?
  - What would they want to accomplish?
- Find weakest link(s)
  - Crypto library is probably pretty good
  - Is there a way to work around crypto?
    - Data stored in encrypted form; where is *key* stored?
- Main point
  - Do security analysis of the whole system
  - Spend your time where it matters

General categories



Program Component

- Avoid buffer overflow
- Secure software design
- Language-specific problems
- Application-specific issues

Validate input

Respond judiciously

Call other's code carefully

Checking secure software

- Many rules for writing secure code
  - “sanitize user input before using it”
  - “check permissions before doing operation X”
- How to find errors?
  - Formal verification
    - + rigorous
    - – costly, expensive. *Very* rare for software
  - Testing:
    - + simple, few false positives
    - – requires running code: doesn’t scale & can be impractical
  - Manual inspection
    - + flexible
    - – erratic & doesn’t scale well.

Two options

- Static analysis
  - Inspect code or run automated method to find errors or gain confidence about their absence

- Dynamic analysis
  - Run code, possibly under instrumented conditions, to see if there are likely problems

Static vs Dynamic Analysis

- Static
  - Consider all possible inputs (in summary form)
  - Find bugs and vulnerabilities
  - Can prove absence of bugs, in some cases

- Dynamic
  - Need to choose sample test input
  - Can find bugs vulnerabilities
  - Cannot prove their absence

Static Analysis

- Abstracts program properties and/or looks for problems
- Tools come from program analysis
  - Type inference, data flow analysis, theorem proving
- Also manual static analysis aka reverse engineering
- Usually on source code, can be on byte code or assembly code
- Strengths
  - Complete code coverage (in theory)
  - Potentially verify absence/report all instances of whole class of bugs
  - Catches different bugs than dynamic analysis
- Weaknesses
  - High false positive rates
  - Many properties cannot be easily modeled
  - Difficult to build
  - Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)

Reverse Engineering (for pentesting)

- A special form of static analysis
  - Expert user required
  - Used to study how a program works and to find vulnerabilities that can be exploited in closed source software
  - Find backdoors (insider attack …)
  - Also used to study malware, and how it exploits software/systems
- Reversing binary into:
  - Assembly form (given file format)
    - Executable and Linkable Format (ELF) – Linux
    - Portable Executable (PE) Format – Windows
    - Mach-Object (Mach-O) – OSX and iOS
    - ART (replacing Dalvik) - Android
  - Source code form (less common, e.g. Java disassembly from bytecode)

Reverse Engineering

- Requires a tool for performing disassembly:
  - IDA (Pro): world famous disassembly tool
  - Ghidra: disassembler authored by NSA
  - Radare2 (Linux/Mac/Windows)
  - Objdump (Linux/Mac)
  - Hopper (Linux)
- May also require using a debugger and an assembler for modifying the assembly code:
  - Ollydbg (Windows)
  - GDB (Linux/Mac)

Disassemblers vs. Debuggers

- Debuggers are designed to run code
  - They can disassemble code (e.g. gdb « disas »)
    - Single functions
    - Based on the instruction pointer
  - Generally don’t do batch disassembly
- Disassemblers don’t run the code
  - Output is a disassembly listing
    - Often quite to extremely large output
    - Hard to navigate
    - Harder to understand than source code!
  - Advanced tools also provide a control-flow graph view with an intuitive navigation
    - And many other tools/functionalities (renaming, reformatting, introducing comments, hexdump, code structure analysis, library analysis)

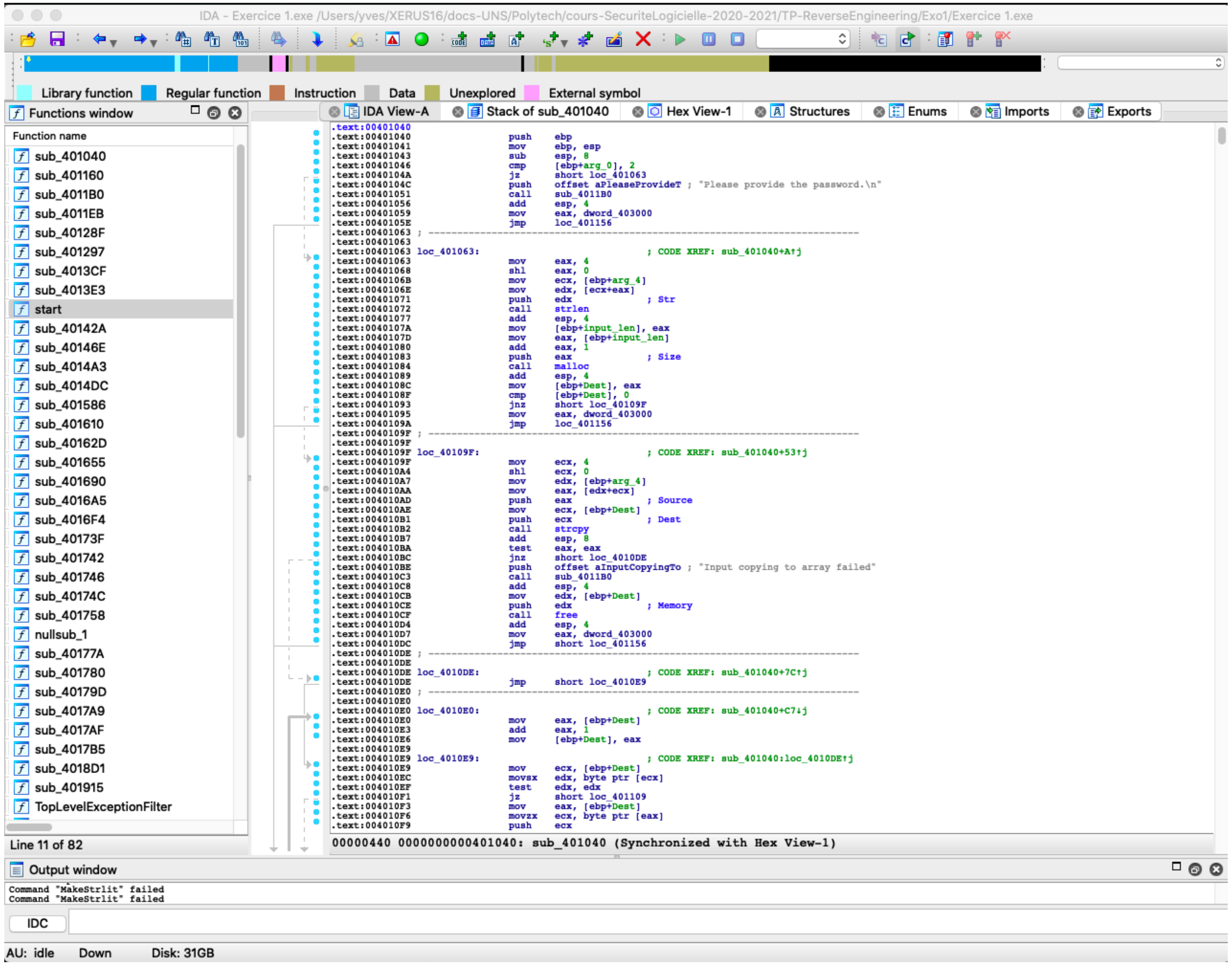
IDA Operation

- Load your binary of interest (e.g. drag&drop)
- IDA analyzes and characterizes each byte of the binary file
  - Builds a database (see files under your directory)
  - Further manipulations will involve database interactions (reads for navigation, updates for renaming, etc.)
- Performs a detailed analysis of the code:
  - Recognizes function boundaries and library calls (and even names for known library calls)
  - Recognizes data types for known library calls
  - Recognizes string constants
- You can navigate code and graph (double-click)
  - Web browser like history (and ESC = back)
- You can modify content as you recognize data and functions (change names)
  - Beware: many hotkeys, and there is no undo!
- May even « execute » code together with a remote debugger
  - Breakpoints can be set within disassembler

IDA: Disassembly listing

- main window
  - initially positioned at entry point
  - Entry point = generally not main, but instead start or _start
- Can switch with graph view using space bar
- Also contains jumps (conditional or not) in the margin at the left of the assembly code dump
  - Useful for identifying branching and looping constructs
  - Conditional jumps – dashed
  - Unconditional jumps – solid
  - Backward jumps – heavier line

IDA: disassembly listing



```
sub_401040
sub_401160
sub_4011B0
sub_4011EB
sub_40128F
sub_401297
sub_4013CF
sub_4013E3
start
sub_40142A
sub_40146E
sub_4014A3
sub_4014DC
sub_401586
sub_401610
sub_40162D
sub_401655
sub_401690
sub_4016A5
sub_4016F4
sub_40173F
sub_401742
sub_401746
sub_40174C
sub_401758
nullsub_1
sub_40177A
sub_401780
sub_40179D
sub_4017A9
sub_4017AF
sub_4017B5
sub_4018D1
sub_401915
TopLevelExceptionFilter
```

Line 11 of 82

```
.text:00401040
.text:00401040                 push    ebp
.text:00401041                 mov     ebp, esp
.text:00401043                 sub     esp, 8
.text:00401046                 cmp     [ebp+arg_0], 2
.text:0040104A                 jz      short loc_401063
.text:0040104C                 push    offset aPleaseProvideT ; "Please provide the password.\n"
.text:00401051                 call    sub_4011B0
.text:00401056                 add     esp, 4
.text:00401059                 mov     eax, dword_403000
.text:0040105E                 jmp     loc_401156
.text:00401063 ; ---------------------------------------------------------------------------
.text:00401063
.text:00401063 loc_401063:                             ; CODE XREF: sub_401040+A↑j
.text:00401063                 mov     eax, 4
.text:00401068                 shl     eax, 0
.text:0040106B                 mov     ecx, [ebp+arg_4]
.text:0040106E                 mov     edx, [ecx+eax]
.text:00401071                 push    edx             ; Str
.text:00401072                 call    strlen
.text:00401077                 add     esp, 4
.text:0040107A                 mov     [ebp+input_len], eax
.text:0040107D                 mov     eax, [ebp+input_len]
.text:00401080                 add     eax, 1
.text:00401083                 push    eax             ; Size
.text:00401084                 call    malloc
.text:00401089                 add     esp, 4
.text:0040108C                 mov     [ebp+Dest], eax
.text:0040108F                 cmp     [ebp+Dest], 0
.text:00401093                 jnz     short loc_40109F
.text:00401095                 mov     eax, dword_403000
.text:0040109A                 jmp     loc_401156
.text:0040109F ; ---------------------------------------------------------------------------
.text:0040109F
.text:0040109F loc_40109F:                             ; CODE XREF: sub_401040+53↑j
.text:0040109F                 mov     ecx, 4
.text:004010A4                 shl     ecx, 0
.text:004010A7                 mov     edx, [ebp+arg_4]
.text:004010AA                 mov     eax, [edx+ecx]
.text:004010AD                 push    eax             ; Source
.text:004010AE                 mov     ecx, [ebp+Dest]
.text:004010B1                 push    ecx             ; Dest
.text:004010B2                 call    strcpy
.text:004010B7                 add     esp, 8
.text:004010BA                 test    eax, eax
.text:004010BC                 jnz     short loc_4010DE
.text:004010BE                 push    offset aInputCopyingTo ; "Input copying to array failed"
.text:004010C3                 call    sub_4011B0
.text:004010C8                 add     esp, 4
.text:004010CB                 mov     edx, [ebp+Dest]
.text:004010CE                 push    edx             ; Memory
.text:004010CF                 call    free
.text:004010D4                 add     esp, 4
.text:004010D7                 mov     eax, dword_403000
.text:004010DC                 jmp     short loc_401156
.text:004010DE ; ---------------------------------------------------------------------------
.text:004010DE
.text:004010DE loc_4010DE:                             ; CODE XREF: sub_401040+7C↑j
.text:004010DE                 jmp     short loc_4010E9
.text:004010E0 ; ---------------------------------------------------------------------------
.text:004010E0
.text:004010E0 loc_4010E0:                             ; CODE XREF: sub_401040+C7↓j
.text:004010E0                 mov     eax, [ebp+Dest]
.text:004010E3                 add     eax, 1
.text:004010E6                 mov     [ebp+Dest], eax
.text:004010E9
.text:004010E9 loc_4010E9:                             ; CODE XREF: sub_401040:loc_4010DE↑j
.text:004010E9                 mov     ecx, [ebp+Dest]
.text:004010EC                 movsx   edx, byte ptr [ecx]
.text:004010EF                 test    edx, edx
.text:004010F1                 jz      short loc_401109
.text:004010F3                 mov     eax, [ebp+Dest]
.text:004010F6                 movzx   ecx, byte ptr [eax]
.text:004010F9                 push    ecx
```

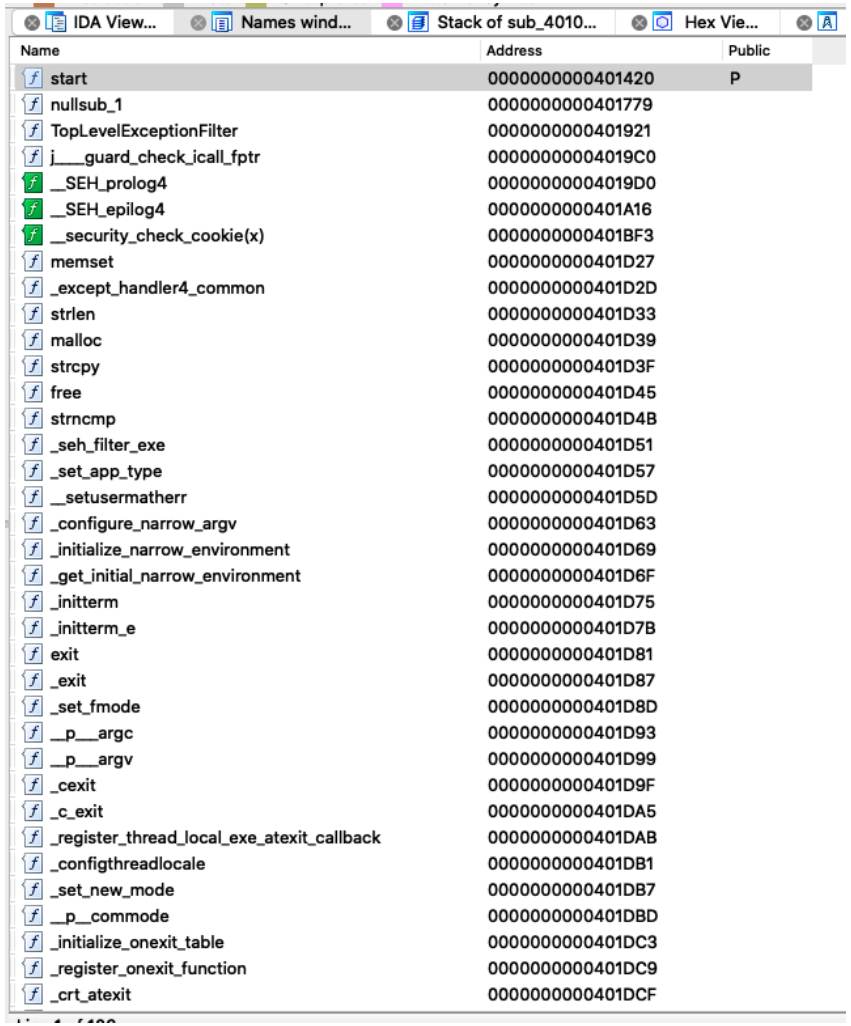
```
00000440 0000000000401040: sub_401040 (Synchronized with Hex View-1)
```

```
Command "MakeStrlit" failed
Command "MakeStrlit" failed
```

IDA: Names window

- Based on imports, exports, and some analysis
  - F is a function
  - L is a library function
  - C is code/instruction
  - A is a string
  - D is defined data
  - I is an imported function (dynamically linked)

IDA: Names window

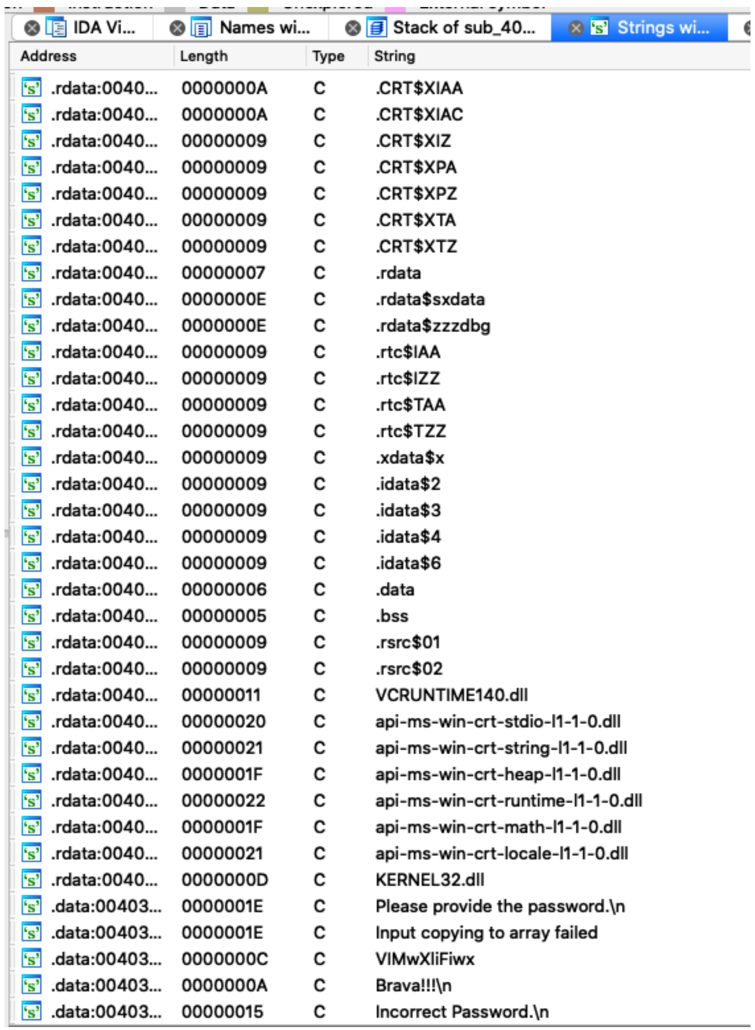


Name	Address	Public
start	0000000000401420	P
nullsub_1	0000000000401779	
TopLevelExceptionFilter	0000000000401921	
j___guard_check_icall_fptr	00000000004019C0	
__SEH_prolog4	00000000004019D0	
__SEH_epilog4	0000000000401A16	
__security_check_cookie(x)	0000000000401BF3	
memset	0000000000401D27	
_except_handler4_common	0000000000401D2D	
strlen	0000000000401D33	
malloc	0000000000401D39	
strcpy	0000000000401D3F	
free	0000000000401D45	
strncmp	0000000000401D4B	
_seh_filter_exe	0000000000401D51	
_set_app_type	0000000000401D57	
__setusermatherr	0000000000401D5D	
_configure_narrow_argv	0000000000401D63	
_initialize_narrow_environment	0000000000401D69	
_get_initial_narrow_environment	0000000000401D6F	
_initterm	0000000000401D75	
_initterm_e	0000000000401D7B	
exit	0000000000401D81	
_exit	0000000000401D87	
_set_fmode	0000000000401D8D	
__p___argc	0000000000401D93	
__p___argv	0000000000401D99	
_cexit	0000000000401D9F	
_c_exit	0000000000401DA5	
_register_thread_local_exe_atexit_callback	0000000000401DAB	
_configthreadlocale	0000000000401DB1	
_set_new_mode	0000000000401DB7	
__p__commode	0000000000401DBD	
_initialize_onexit_table	0000000000401DC3	
_register_onexit_function	0000000000401DC9	
_crt_atexit	0000000000401DCF	

IDA: String window

- Complete listing of strings embedded within the program
- Configurable
  - Right click in Strings window and choose setup
  - Can change minimum length or style of string to search for (IDA rescans for strings if you change settings)
- Excellent tool for locating interesting inputs/outputs or text data
  - E.g., detect success conditions in the code

IDA: String window

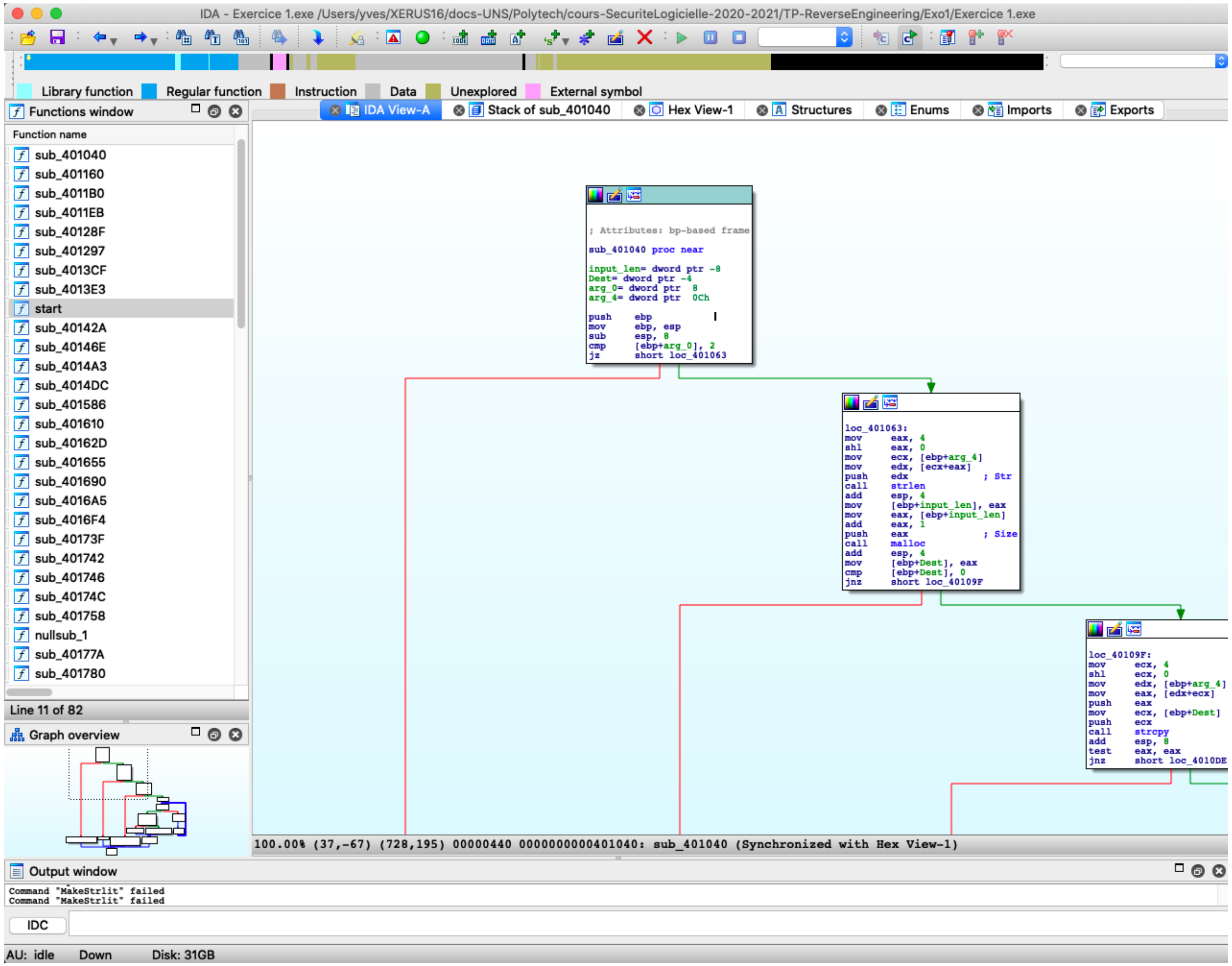


Address	Length	Type	String
.rdata:0040...	0000000A	C	.CRT$XIAA
.rdata:0040...	0000000A	C	.CRT$XIAC
.rdata:0040...	00000009	C	.CRT$XIZ
.rdata:0040...	00000009	C	.CRT$XPA
.rdata:0040...	00000009	C	.CRT$XPZ
.rdata:0040...	00000009	C	.CRT$XTA
.rdata:0040...	00000009	C	.CRT$XTZ
.rdata:0040...	00000007	C	.rdata
.rdata:0040...	0000000E	C	.rdata$sxdata
.rdata:0040...	0000000E	C	.rdata$zzzdbg
.rdata:0040...	00000009	C	.rtc$IAA
.rdata:0040...	00000009	C	.rtc$IZZ
.rdata:0040...	00000009	C	.rtc$TAA
.rdata:0040...	00000009	C	.rtc$TZZ
.rdata:0040...	00000009	C	.xdata$x
.rdata:0040...	00000009	C	.idata$2
.rdata:0040...	00000009	C	.idata$3
.rdata:0040...	00000009	C	.idata$4
.rdata:0040...	00000009	C	.idata$6
.rdata:0040...	00000006	C	.data
.rdata:0040...	00000005	C	.bss
.rdata:0040...	00000009	C	.rsrc$01
.rdata:0040...	00000009	C	.rsrc$02
.rdata:0040...	00000011	C	VCRUNTIME140.dll
.rdata:0040...	00000020	C	api-ms-win-crt-stdio-l1-1-0.dll
.rdata:0040...	00000021	C	api-ms-win-crt-string-l1-1-0.dll
.rdata:0040...	0000001F	C	api-ms-win-crt-heap-l1-1-0.dll
.rdata:0040...	00000022	C	api-ms-win-crt-runtime-l1-1-0.dll
.rdata:0040...	0000001F	C	api-ms-win-crt-math-l1-1-0.dll
.rdata:0040...	00000021	C	api-ms-win-crt-locale-l1-1-0.dll
.rdata:0040...	0000000D	C	KERNEL32.dll
.data:00403...	0000001E	C	Please provide the password.\n
.data:00403...	0000001E	C	Input copying to array failed
.data:00403...	0000000C	C	VIMwXliFiwx
.data:00403...	0000000A	C	Brava!!!\n
.data:00403...	00000015	C	Incorrect Password.\n

IDA: graphs

- Many graphs can be generated
- Function flow charts
  - Unconditional jumps – blue line
  - Conditional jump if true – green line
  - Conditional jump if false – red line
  - Move your mouse on top of graph to get further info
- Function call tree (forest) for a program
- All crossrefs from a function (« which other functions do I call? »)
- All crossrefs to a function (« who calls function? »)

IDA: function flow chart



IDA - Exercice 1.exe /Users/yves/XERUS16/docs-UNS/Polytech/cours-SecuriteLogicielle-2020-2021/TP-ReverseEngineering/Exo1/Exercice 1.exe

Function name

- sub_401040
- sub_401160
- sub_4011B0
- sub_4011EB
- sub_40128F
- sub_401297
- sub_4013CF
- sub_4013E3
- start
- sub_40142A
- sub_40146E
- sub_4014A3
- sub_4014DC
- sub_401586
- sub_401610
- sub_40162D
- sub_401655
- sub_401690
- sub_4016A5
- sub_4016F4
- sub_40173F
- sub_401742
- sub_401746
- sub_40174C
- sub_401758
- nullsub_1
- sub_40177A
- sub_401780

Line 11 of 82

```
; Attributes: bp-based frame

sub_401040 proc near

input_len= dword ptr -8
Dest= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+arg_0], 2
jz      short loc_401063
```

```
loc_401063:
mov     eax, 4
shl     eax, 0
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+eax]
push    edx             ; Str
call    strlen
add     esp, 4
mov     [ebp+input_len], eax
mov     eax, [ebp+input_len]
add     eax, 1
push    eax             ; Size
call    malloc
add     esp, 4
mov     [ebp+Dest], eax
cmp     [ebp+Dest], 0
jnz     short loc_40109F
```

```
loc_40109F:
mov     ecx, 4
shl     ecx, 0
mov     edx, [ebp+arg_4]
mov     eax, [edx+ecx]
push    eax
mov     ecx, [ebp+Dest]
push    ecx
call    strcpy
add     esp, 8
test    eax, eax
jnz     short loc_4010DE
```

```
100.00% (37,-67) (728,195) 00000440 0000000000401040: sub_401040 (Synchronized with Hex View-1)
```

```
Command "MakeStrlit" failed
Command "MakeStrlit" failed
```

AU: idle   Down   Disk: 31GB

x86 Assembly basics: instructions

- Memory manipulation:
  - Mov <dst>, <src>   - adresses can be described by « [address value] »
  - Push/Pop <registry>
  - Xcgh <registry 1>, <registry 2>
- Arithmetic operators:
  - Add/Dec/Mul/Div <registry>, <operand2>
  - Inc/Dec <registry>
  - Neg <registry> - two’s complement
- Bit-level manipulation:
  - And/Or/Xor <registry1>, <registry2>
  - Not <registry>
  - Shl <registry>, <added_bit>
  - Shr <registry>
  - Rol/Ror <registry>

x86 Assembly basics: instructions

- Tests
  - Cmp <registry 1>, <registry 2>
- Jumps
  - Jmp <code location>
  - Je <code location> (if previous test is equal)
  - Jne <code location> (if previous test not equal)
  - Jz <code location> (if previous operation is zero)
  - Jnz <code location> (if previous operation not zero)
- Subroutine calls
  - Call <code location>
  - Ret

Two Types of Tool Based Static Analysis

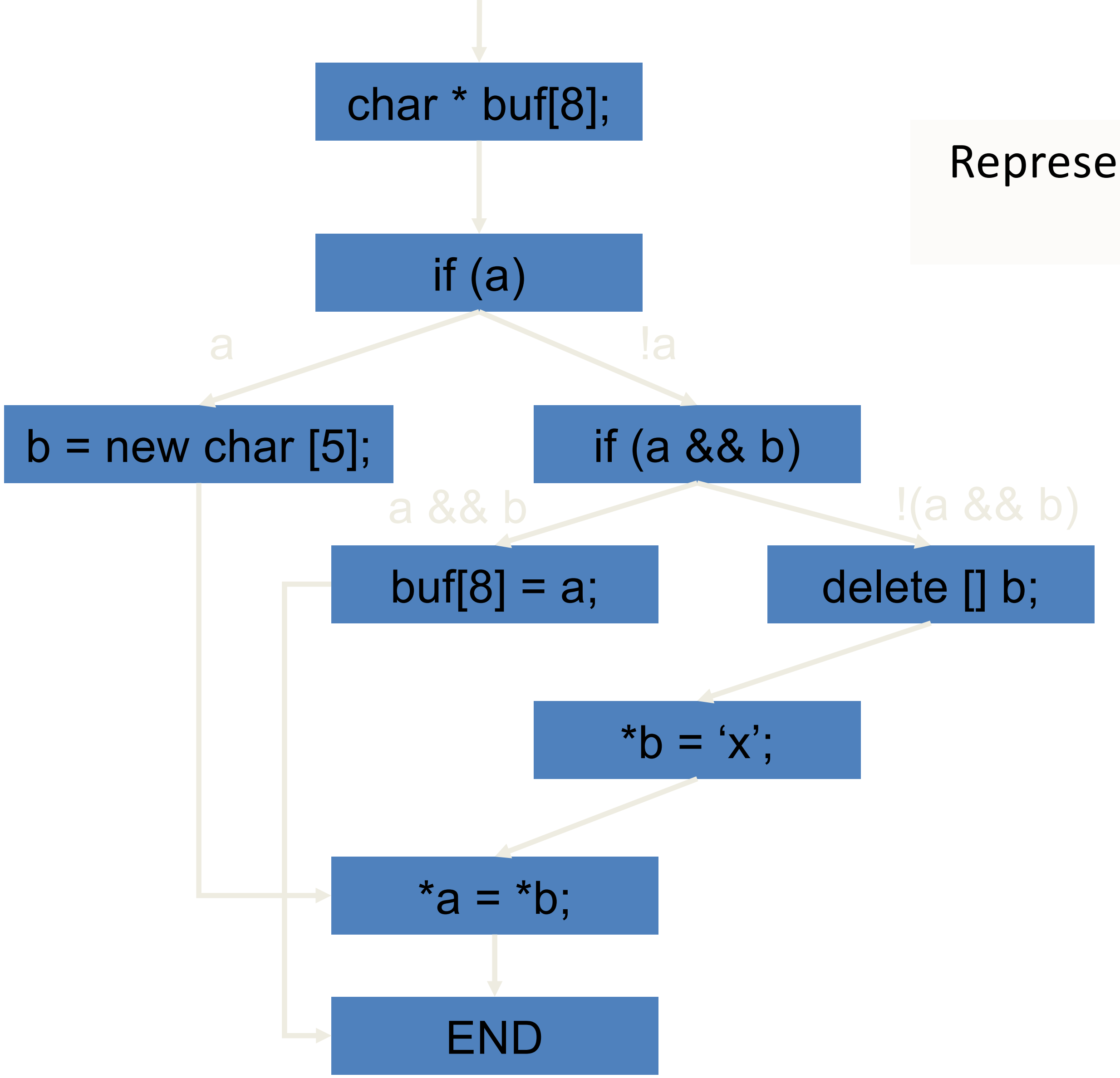
- (Rather) Shallow code analysis.
  - Look for known code issues: e.g., unsafe string functions `strncpy(), sprintf(), gets()`
  - Look for unsafe functions in your source base
  - Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)
- Deeper analysis
  - Requires complex code parsing and computations
  - Some are implemented in tools like coverity, fortify, visual studio …
  - Otherwise must be developed on top of parsers like LLVM
  - In the case of disassemblers, the security expert is the last part of the static analyzer …

Static analysis: Soundness, Completeness

Property	Definition
Soundness	“Sound for reporting correctness” Analysis finds a bug → There is a bug
Completeness	“Complete for reporting correctness” No bug → Analysis says no bug

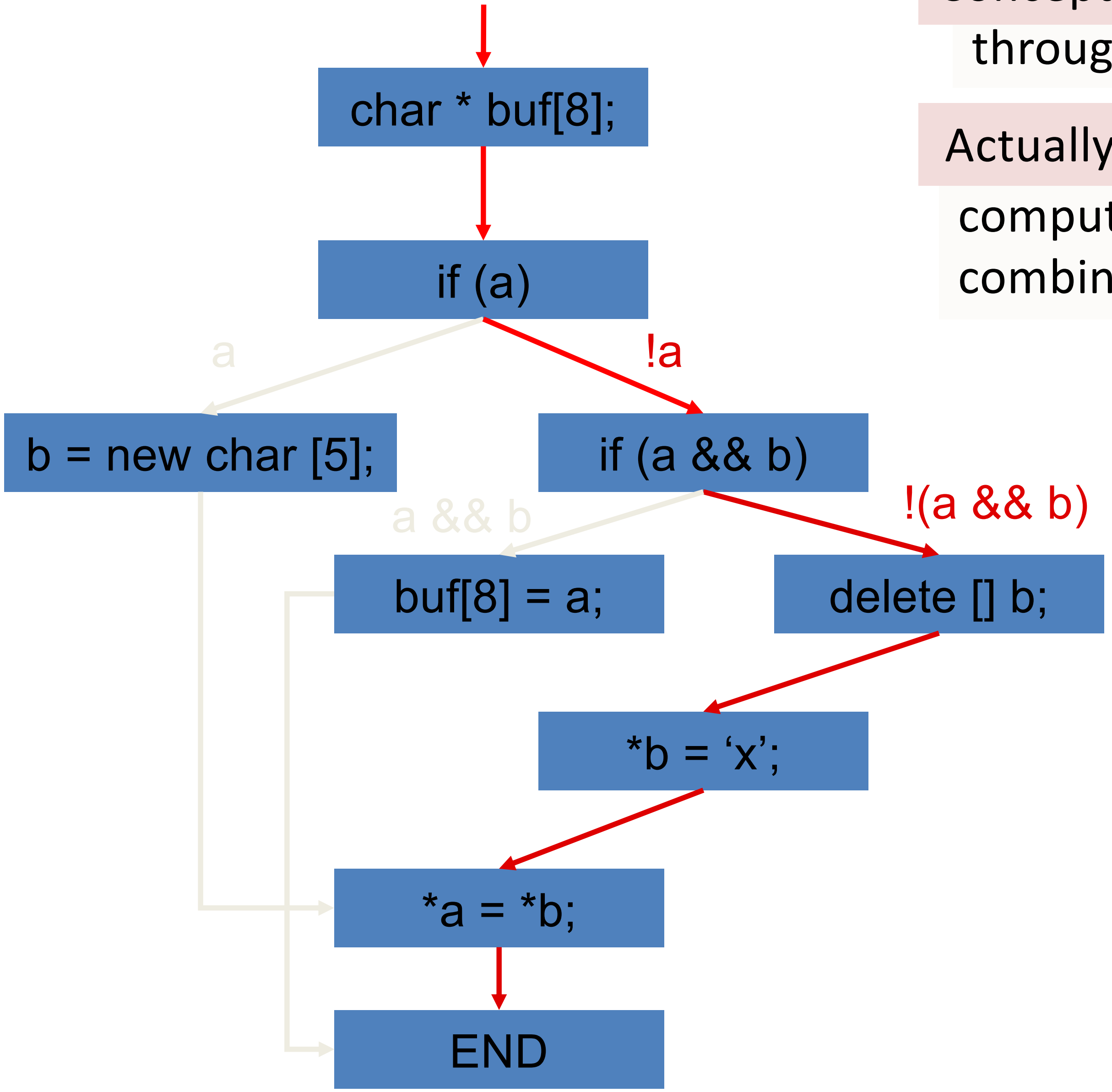
	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	May not report all errors Reports no false alarms **Decidable**
Unsound	Reports all errors May report false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Control Flow Graph (CFG)



Represent logical structure of code in graph form

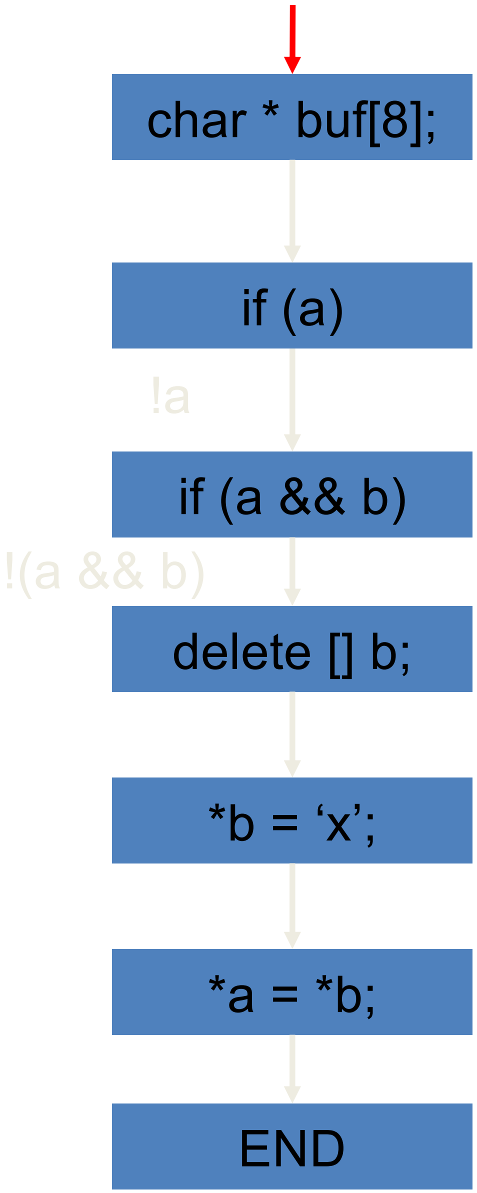
Path Traversal



Conceptually Analyze each path through control graph separately

Actually Perform some checking computation once per node; combine paths at merge nodes

Apply Checking



char * buf[8];

if (a)

!a

if (a && b)

!(a && b)

delete [] b;

*b = ‘x’;

*a = *b;

END

three checkers can be run for this path

Null pointers Use after free Array overrun

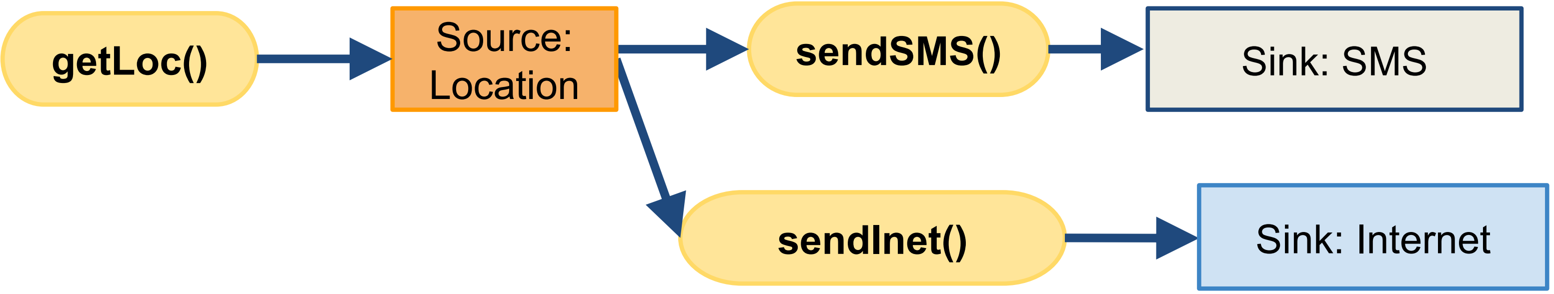
Checker

- Defined by a state diagram, with state transitions and error states

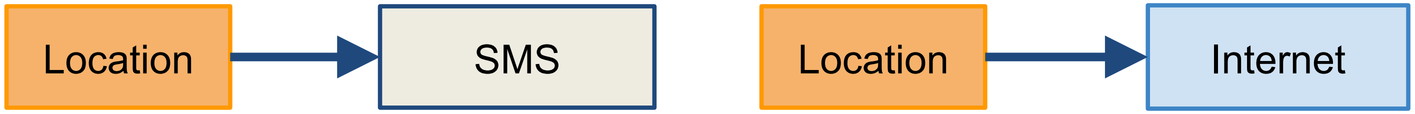
Run Checker

- Assign initial state to each program variable
- State at program point depends on state at previous point + program actions
- Emit error if error state reached

Data Flow Analysis

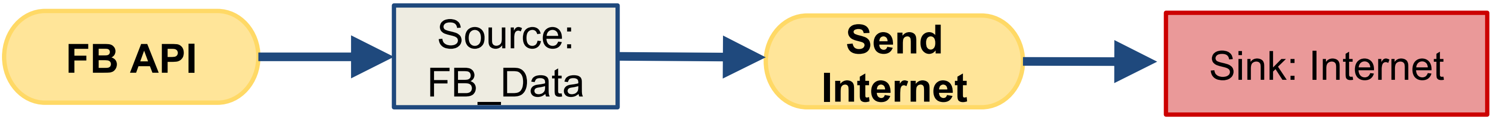


- Source-to-sink flows
    - Sources: Location, Calendar, Contacts, Device ID etc.
    - Sinks: Internet, SMS, Disk, etc.



Applications of Data Flow Analysis

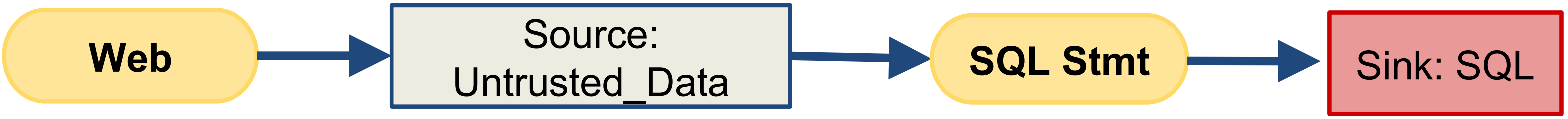
- Vulnerability Discovery
- Malware/Greyware Analysis
  - Data flow summaries enable enterprise-specific policies
- API Misuse and Data Theft Detection



- Automatic Generation of App Privacy Policies
  - Avoid liability, protect consumer privacy

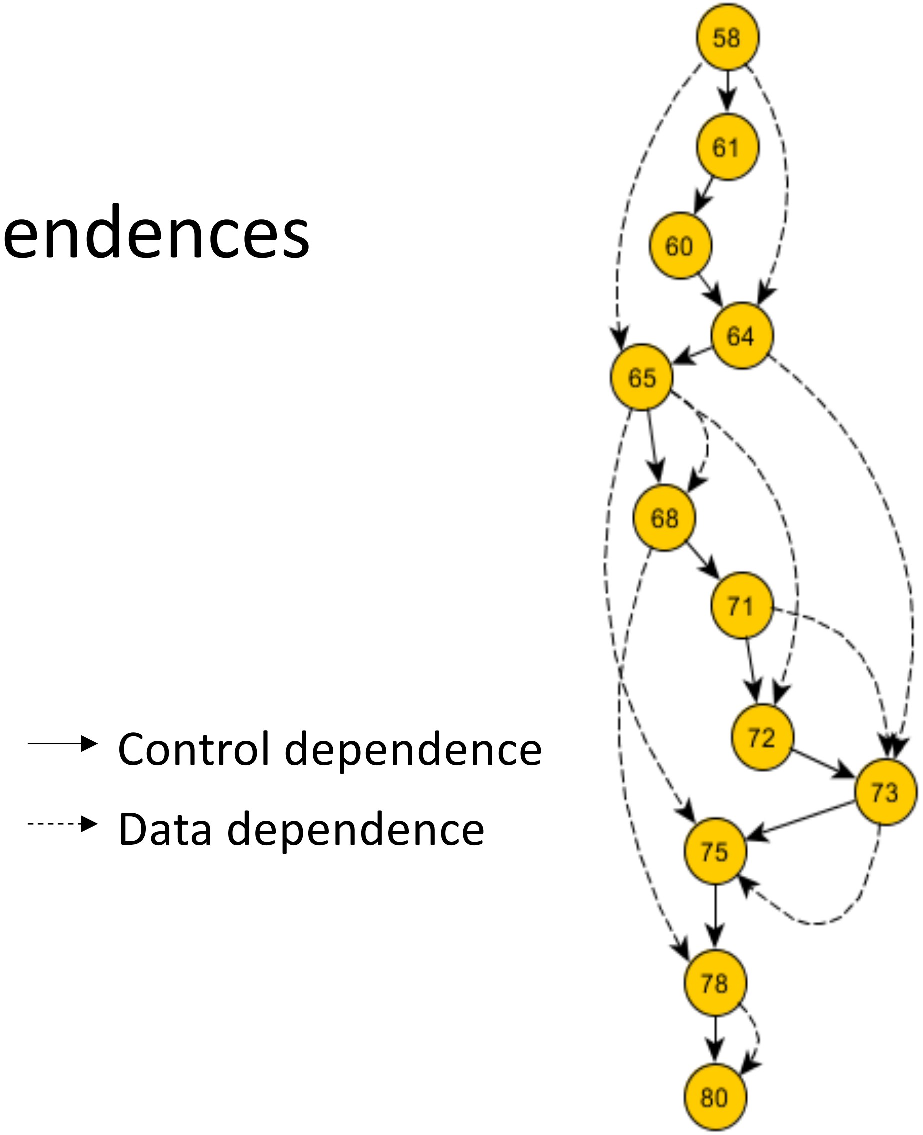
Privacy Policy

This app collects your:
Contacts
Phone Number
Address



Program Dependence Graph (PDG)

- Control Dependences
- Explicit + Implicit Data Dependences
- Properties:
  - Path-sensitive
  - Context-Sensitive
  - Object-Sensitive

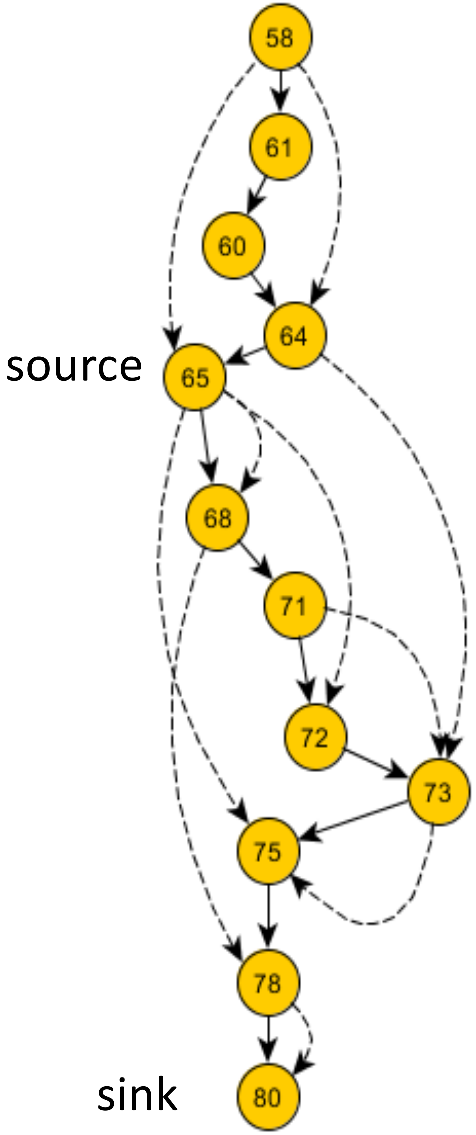


→ Control dependence

⇢ Data dependence

JOANA IFC tool

- Intended for Information Flow Analysis
- Annotations: SINK / SOURCE
- Non-Interference: Security Levels (HIGH / LOW)


58
61
60
64
source
65
68
71
72
73
75
78
sink
80