Interprocedural Analysis

Basic Concepts and Why?

Marek Fešar
Tomáš Fiedor

Brno University of Technology
Faculty of Information Technology

2. prosince 2012

Why bother?

- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality

Why bother?

- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality

- Still, the question is: Why should we bother?!

Why bother?

- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality

- Still, the question is: Why should we bother?!
    - Optimizations (Constant Propagation, Memory Management, etc.)

Why bother?

- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality
- Still, the question is: Why should we bother?!
    - Optimizations (Constant Propagation, Memory Management, etc.)
    - Vulnerability Detection (SQL Injection, Buffer Overflow)

Why bother?

- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality
- Still, the question is: Why should we bother?!
    - Optimizations (Constant Propagation, Memory Management, etc.)
    - Vulnerability Detection (SQL Injection, Buffer Overflow)
    - Further Code Analysis (Virtual Methods, Pointer Analysis, etc.)

Why bother?

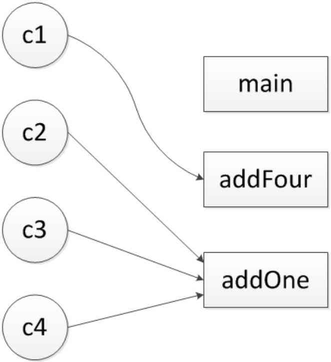
- Most of the programs are complex
    - Not just big chunk of code
    - Lots of cooperating functions
    - Each fuction has impact on program functionality
- Still, the question is: Why should we bother?!
    - Optimizations (Constant Propagation, Memory Management, etc.)
    - Vulnerability Detection (SQL Injection, Buffer Overflow)
    - Further Code Analysis (Virtual Methods, Pointer Analysis, etc.)
    - and most of all, it is

Why bother?

- Most of the programs are complex
  - Not just big chunk of code
  - Lots of cooperating functions
  - Each fuction has impact on program functionality
- Still, the question is: Why should we bother?!
  - Optimizations (Constant Propagation, Memory Management, etc.)
  - Vulnerability Detection (SQL Injection, Buffer Overflow)
  - Further Code Analysis (Virtual Methods, Pointer Analysis, etc.)
  - and most of all, it is FUN

Basic notations

- **Call Graph** = representation of function calls:



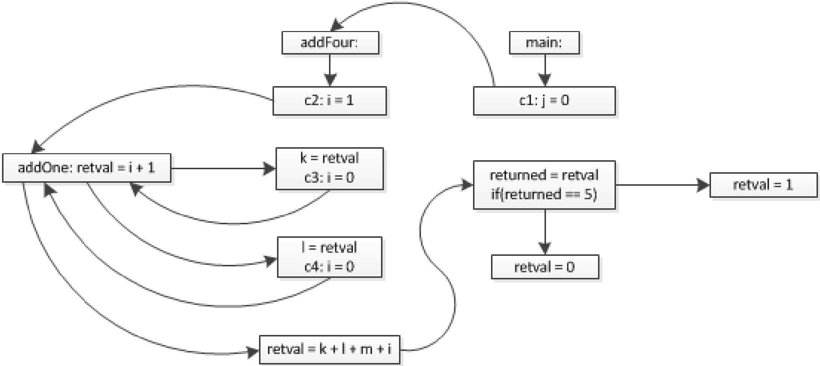
```
   int main() {
c1:  returned = addFour(0);
     if(returned == 5) {
       return 1;
     }
     return 0;
   }

   int addOne(int i) {
     return i+1;
   }

   int addThree(int j) {
c2:   int k = addOne(1);
c3:   int l = addOne(0);
c4:   int m = addOne(0);
      return j+k+l+m;
   }
```

Context-insenstive = Super Control-graphs!

- Introducing new edges → from call sites and back to code

Context-insenstive = Super Control-graphs!

- Introducing new edges → from call sites and back to code



Call strings

- Context can be described by its call site
- **Call string** = part of the call stack contents
- ***k*-limiting context analysis** = only *k* most immediate call sites create the call string

```
int main() {
c1: returned = addFour(0);{
    if(returned == 5) {
       return 1;
    }
    return 0;
  }

  int addOne(int i) {
    return i+1;
  }

  int addThree(int j) {
c2:  int k = addOne(1);
c3:  int l = addOne(0);
c4:  int m = addOne(0);
     return j+k+l+m;
  }
```

Call strings

- Context can be described by its call site
- **Call string** = part of the call stack contents
- ***k*-limiting context analysis** = only *k* most immediate call sites create the call string
- So we get these call strings:
    - (c1, c2)
    - (c1, c3)
    - (c1, c4)

```
int main() {
c1: returned = addFour(0);{
      if(returned == 5) {
         return 1;
      }
      return 0;
   }

   int addOne(int i) {
      return i+1;
   }

   int addThree(int j) {
c2:   int k = addOne(1);
c3:   int l = addOne(0);
c4:   int m = addOne(0);
      return j+k+l+m;
   }
```

Cloning Based

- For every function call, one unique context, one unique function
- No confusion, loads of code.

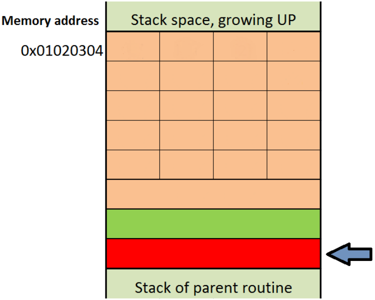
```
    int main() {
c1: returned = addFour(0);
      if(returned == 5)
         return 1;
      return 0;
    }

    int addOne1(int i) {
      return i+1;
    }

    int addOne2(int i) {
      return i+1;
    }

    int addOne3(int i) {
      return i+1;
    }

    int addThree(int j) {
c2:   int k = addOne1(1);
c3:   int l = addOne2(0);
c4:   int m = addOne3(0);
      return j+k+l+m;
    }
```

Summary-Based Analysis

- Summarization of function behaviour with transfer/flow function
- Minimum of function body analysis
- Can be simplified by merging caller informations by meet operator
- After summary-analysis we clone functions on its basis

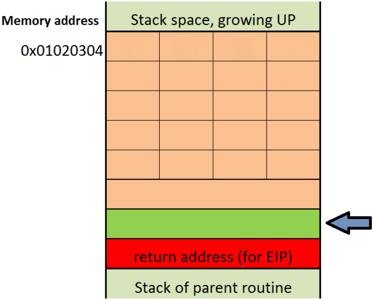
```
    int main() {
c1: returned = addFour(0);
      if(returned == 5) {
         return 1;
      }
      return 0;
    }

    int addOne1(int i) {
      return i+1;
    }

    int addOne0(int i) {
      return i+1;
    }

    int addThree(int j) {
c2:    int k = addOne1(1);
c3:    int l = addOne0(0);
c4:    int m = addOne0(0);
       return j+k+l+m;
    }
```

Snippet of vulnerable code



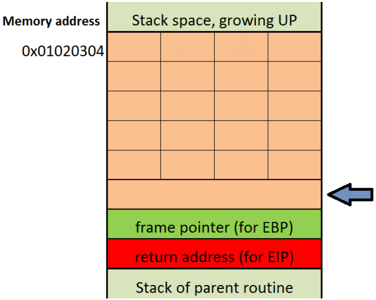
```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



Memory address

0x01020304

Stack space, growing UP

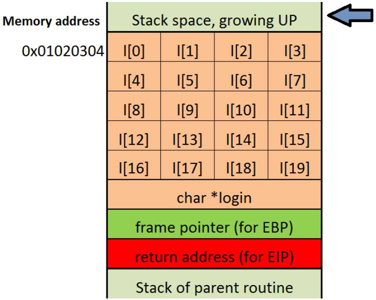
frame pointer (for EBP)

return address (for EIP)

Stack of parent routine

```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



Memory address

0x01020304

Stack space, growing UP

l[0]	l[1]	l[2]	l[3]
l[4]	l[5]	l[6]	l[7]
l[8]	l[9]	l[10]	l[11]
l[12]	l[13]	l[14]	l[15]
l[16]	l[17]	l[18]	l[19]

char *login

frame pointer (for EBP)

return address (for EIP)

Stack of parent routine

```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Injected Shellcode

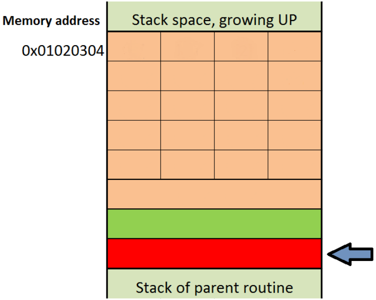
- Lets have following shell code:

```
xor         eax,eax
mov         dx,9998     ; 0x270e
sub         dx,9990     ; 0x2706
mov         al, 55      ; 0x37
int         0x80
```

- After compilation with **NASM** we get:

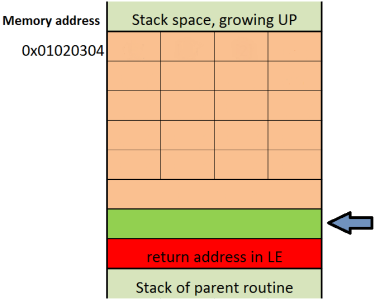
```
char shellcode[] =
    "\x31\xc0\x66\xba\x0e\x27\x66\x81\xea\x06\x27\xb0\x37\xcd\x80";
```

Snippet of vulnerable code



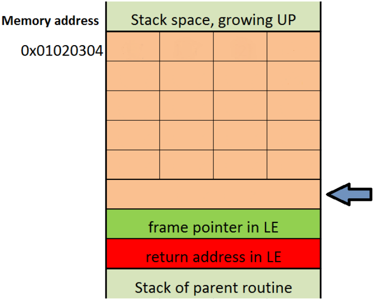
```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



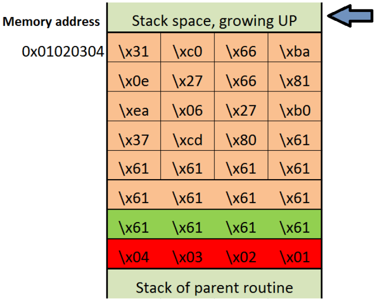
```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

Snippet of vulnerable code



Memory address	Stack space, growing UP			
0x01020304	\x31	\xc0	\x66	\xba
	\x0e	\x27	\x66	\x81
	\xea	\x06	\x27	\xb0
	\x37	\xcd	\x80	\x61
	\x61	\x61	\x61	\x61
	\x61	\x61	\x61	\x61
	\x61	\x61	\x61	\x61
	\x04	\x03	\x02	\x01
	Stack of parent routine			

```
void logMeIn(char *login) {
    char l[20];
    // no bounds checked
    strcpy(l, login);
    // .. some more unimportant code
}

int main(int argc, char **argv) {
    logMeIn(argv[1]);
    // ...
}
```

So, what can we do against this?

1. Analyse user input (it is potentionaly dangerous!)

So, what can we do against this?

1. Analyse user input (it is potentionaly dangerous!)
2. Heuristic tools that check if at least some test before dangerous operation (like `strcpy`, `puts`, etc.) takes place

So, what can we do against this?

1. Analyse user input (it is potentionaly dangerous!)
2. Heuristic tools that check if at least some test before dangerous operation (like `strcpy`, `puts`, etc.) takes place
3. Dynamic bounds checking (with help of static analysis)

So, what can we do against this?

1. Analyse user input (it is potentionaly dangerous!)
2. Heuristic tools that check if at least some test before dangerous operation (like `strcpy`, `puts`, etc.) takes place
3. Dynamic bounds checking (with help of static analysis)
4. Bound checking insertion (complex)

What should we remember?

- Interprocedural analysis is useful in many ways:
  - Optimizations
  - Further analysis
  - Bug hunting

What should we remember?

- Interprocedural analysis is useful in many ways:
  - Optimizations
  - Further analysis
  - Bug hunting
- Several basic approaches:
  - Context Insensitive
  - Call Strings
  - Cloning Based
  - Summary Based

What should we remember?

- Interprocedural analysis is useful in many ways:
    - Optimizations
    - Further analysis
    - Bug hunting
- Several basic approaches:
    - Context Insensitive
    - Call Strings
    - Cloning Based
    - Summary Based
- Buffer overflow = third most dangerous code weakness.

Any questions?

Used/Additional Literature

AHO, A.; LAM, M.; SETHI, R.; et al.: Compilers: Principles, Techniques and Tools. Pearson Education, 2005, ISBN 978-0321486813.

KHEDKER, U.; SANYAL, A.; KARKARE, B.: Data Flow Analysis: Theory and Practice. CRC Press, 2009, ISBN 978-0848328800.

The MITRE Corporation.: Common Weakness Enumeration – CWE-120: Buffer Copy Without Checking Size of Input ('Classic Buffer Overflow'). [online], Last Updated 14.5.2012, [cit. 2012-11-30]. Available at: http://cwe.mitre.org/data/definitions/120.html.