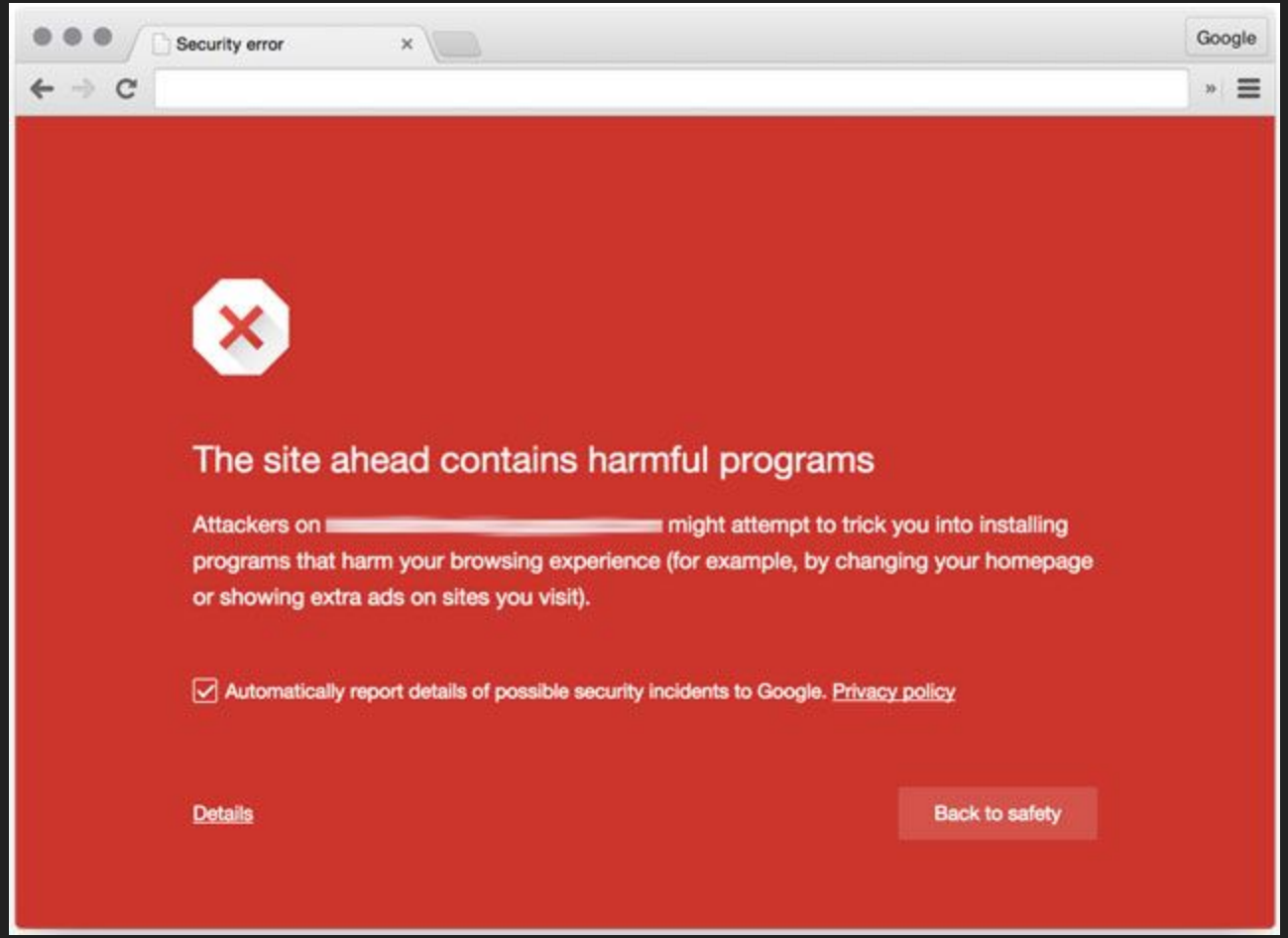
Fuzzing and Browser Exploits

MST - 2/23


Security error
Google
The site ahead contains harmful programs
Attackers on [redacted] might attempt to trick you into installing programs that harm your browsing experience (for example, by changing your homepage or showing extra ads on sites you visit).
Automatically report details of possible security incidents to Google. Privacy policy
Details
Back to safety

Memory Corruption

- **Memory Corruption** attacks occur when we can write somewhere to memory we aren't supposed to
- We can exploit **memory structures** to change the flow of the program
- Methods we've discussed
  - Shellcoding
  - ret2libc
  - ROP
- What else is there?
  - Lots!
  - … but we won't get into them :(

Other (common) forms of Memory Corruption

- Heap overflows
- Integer overflows
- Format string attacks
- Use-after-free
- Double free

Integer Overflows

- Huge or negative numbers are passed in and improperly handled
- Often used to bypass checks or to write memory past the bounds of a buffer
- What's wrong with this program?

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
     int num = atoi(argv[1]);
     char *input = malloc(4 * num);
     for(int x = 0; x < num; x++) {
          input[x] = x;
          input[x+1] = x;
          input[x+2] = x;
          input[x+3] = x;
     }
     /* do some other stuff */
     return 0;
}
```

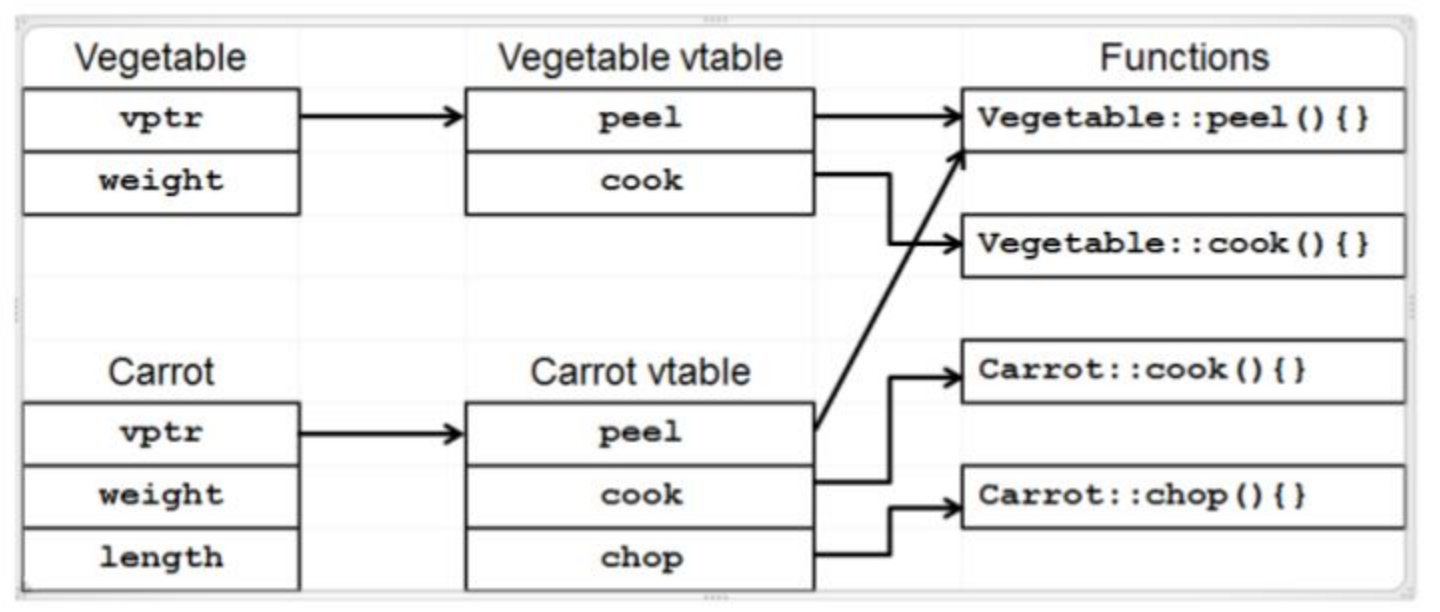
Heap overflows

- Overflowing a data structure, but on the heap instead of the stack
- Heap is for dynamically allocated memory
  - eg. `malloc(100)` goes on the heap
- We can overwrite from one heap object to another nearby one and change the contents of data

	Our Array						(unused)				Target				
	C	Y	R	U	S						H	E	L	L	O

	Our Array						(unused)				Target				
	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	O

Use-after-free



NCC Group, *Exploiting CVE-2014-0282*

Double Free

- A pointer's contents may change after it has been freed
- Calling free on corrupted heap metadata can cause your program to crash
- Calling free on attacker-controlled heap metadata can cause your program to be exploited
- Set your freed pointers to null

```
#include <stdio.h>
#include <unistd.h>
#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
     char *buf1;
     char *buf2;
     char *buf3;
     buf1 = (char *) malloc(BUFSIZE2);
     buf2 = (char *) malloc(BUFSIZE2);
     free(buf1);
     free(buf2);
     buf3 = (char *) malloc(BUFSIZE1);
     strncpy(buf3, argv[1], BUFSIZE1-1);
     free(buf2);
     free(buf3);
}
```

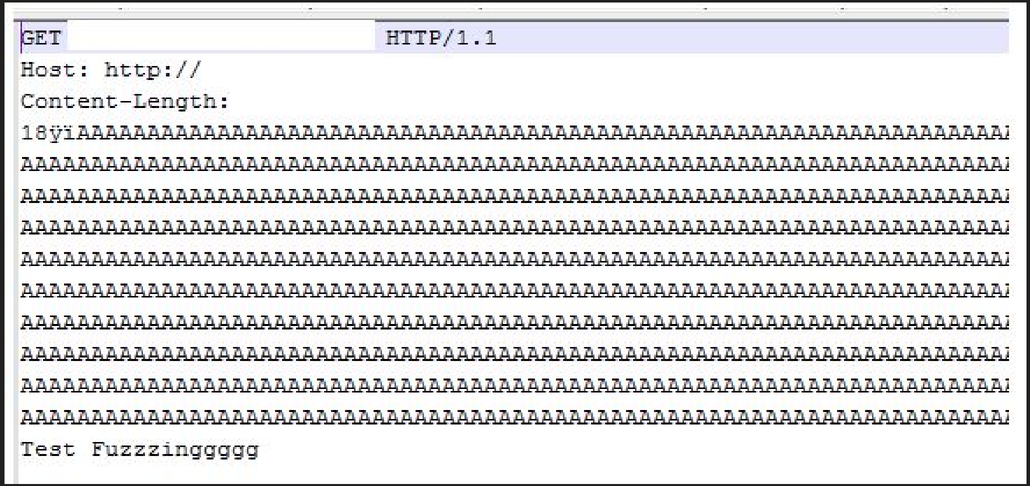
https://cwe.mitre.org/data/definitions/415.html

Fuzzing

- Security and binary defenses are improving
  - Time + money investment to find a single bug is increasing
- Exploitation is (usually) straightforward -- finding the bug is hard
- How can we efficiently test large software with thousands of functions?
  - It takes long enough for even small programs...
- **Fuzzing** lets us test software automatically for vulnerabilities
  - Responsible for **overwhelming majority** of all new software vulnerabilities found in major products

Fuzzing

1. Take a series of valid program inputs
2. For each input string in the list, perform **random mutations** on it to generate a new input
3. Send the new input to the program and see if it crashes
4. Repeat forever!



Fuzzing

1. Take a series of valid program inputs
2. For each input string in the list, perform **random mutations** on it to generate a new input
3. Send the new input to the program and see if it crashes
4. Repeat forever!

```
void do_things(int which) {
    void (*call_me)();
    switch(which) {
        case 1:
            call_me = foo;
            break;
        case 2:
            call_me = bar;
            break;
    }
    call_me();
}
```

“Dumb” Fuzzers

- No concept of data format, only as an input string
- No setup or prior-knowledge of the target involved
- Randomly mutations some known inputs
  - Flip bits
  - Flip bytes
  - Switch bytes
  - Add or remove random data

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char buffer[10] = {0};
    read(STDIN, buffer, 9);
    if (buffer[3] != 'q') {
        crash_and_burn();
    }
    print(“Ok!\n”);
    return 0;
}

$ cat “AAAqAAAAA” | ./weird
Ok!
$ cat “AAAAAAAAA” | ./weird
crash and burn?
```

“Dumb” Fuzzers

- Very slow for **structured data** (Which is most data)
  - File formats, network protocols, data with checksums, etc
- We spent most of the time stuck on the highlighted section, so we may never find the crash!

```
#include <stdio.h>
int main(int argc, char *argv[]) {
     char buffer[10] = {0};
     read(STDIN, buffer, 9);
     if (sum(buffer, 10) != 334) {
          return 0;
     }
     if (buffer[3] != 'q') {
          crash_and_burn();
     }
     print(“Ok!\n”);
     return 0;
}
```

Protocol Fuzzers

- Randomly mutate data within the given data structure
- Given a data format, generate **conformant** random input for the program
- Requires **prior knowledge** of the specific file format to work
- Can better target file parsers, network servers/clients, etc

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char buffer[10] = {0};
    read(STDIN, buffer, 9);
    if (sum(buffer, 10) != 334) {
        return 0;
    }
    if (buffer[3] != 'q') {
        crash_and_burn();
    }
    print("Ok!\n");
    return 0;
}
```

Protocol Fuzzers

- We still have serious problems
  - We can miss bugs that are valid but **extremely uncommon**
  - We don’t know **how much** of the program we have targeted
  - We need a **robust and complete format definition** to even get started

Evolutionary Fuzzing (AFL)

- Add instrumentation to the program to **track code coverage**
  - Track program branches
- **Keep** input that increases our code coverage and randomly mutate that
- **Discard** input that doesn't improve coverage

```
#include <stdio.h>
int main(int argc, char *argv[]) {
     char buffer[10] = {0};
     read(STDIN, buffer, 9);
     if (buffer[0] == 'a') {
          if (buffer[1] == 'b') {
               if (buffer[2] == 'c') {
                    die_hard();
               }
          }
     }
     print("Ok!\n");
     return 0;
}
```

Evolutionary Fuzzing (AFL)

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]
Adobe Flash / PCRE [1] [2] [3] [4]	sqlite [1] [2] [3] [4]...	OpenSSL [1] [2] [3] [4] [5] [6] [7]
LibreOffice [1] [2] [3] [4]	poppler [1]	freetype [1] [2]
GnuTLS [1]	GnuPG [1] [2] [3] [4]	OpenSSH [1] [2] [3]
PuTTY [1] [2]	ntpd [1]	nginx [1] [2] [3]
bash (post-Shellshock) [1] [2]	tcpdump [1] [2] [3] [4] [5] [6] [7] [8] [9]	JavaScriptCore [1] [2] [3] [4]

Problems with fuzzing

- Extremely slow for large programs
- We can't test know if we've completely tested all possible code
  - We can *try* to reach all code, but that is not the same
- We struggle with checks like `(x == y && y == 0x12345)` that require specific input values to bypass
  - If x and y are 4-byte ints, AFL will try all 2 * 2^32 attempts
  - This is known as the **state explosion** problem
- We gain no information about program meaning

The Future: SAGE

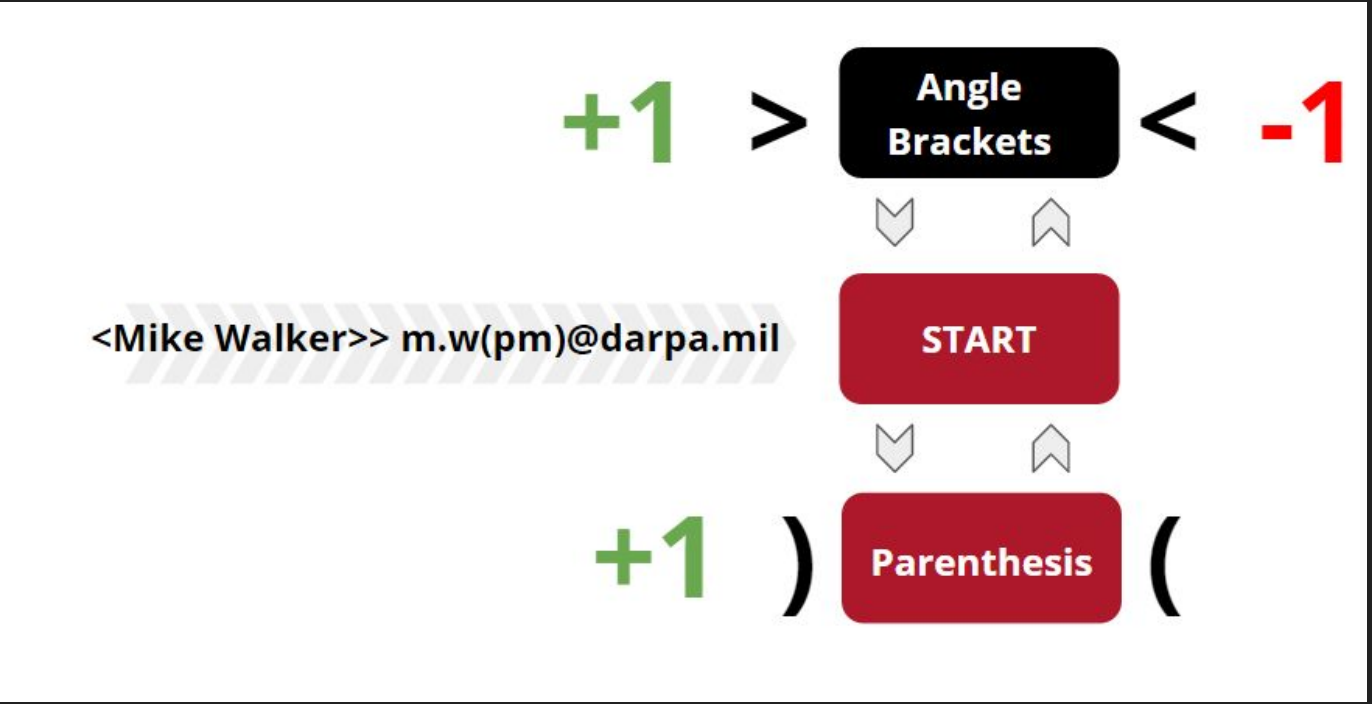
- Microsoft Research project to incorporate symbolic evaluation and fuzzing
  - “Scalable, Automated, Guided Execution”
- First true “smart” fuzzing
- Analyze the program to find **constraints** on input that drive the fuzzing engine
  - Significant reduction in search space
- Within those constraints, fuzz away!
- Code coverage as measure of progress
- Responsible for ⅓ **of pre-release Windows 7 bugs**
- https://www.microsoft.com/en-us/springfield/

```
#include <stdio.h>
int main(int argc, char *argv[]) {
     int buf[3] = {0};
     read(STDIN, buf, 12);
     if (buf[0] + buf[1] = 993) {
          if (buf[1] - 54 = buf[3]) {
               if (sum(buf) = 82) {
                    die_hard();
               }
          }
     }
     print(“Ok!\n”);
     return 0;
}
```

The Future: The Cyber Grand Challenge

- DARPA competition to advance the field of fuzzing and program analysis
- 7 finalists developed systems that combined fuzzing, symbolic execution, static analysis, etc
- Let to some advances in the state of the art
- UVa team placed 2nd!

crackaddr



Source: Trail of Bits

- **201 iterations** through a loop before the bug has an impact
- 10^{65535} possible function inputs, only 1 is vulnerable
- **CGC software found the bug!**

Hack Firefox

Adobe Flash CVE-2014-0556

```
public function copyPixelsToByteArray(rect:Rectangle, data:ByteArray) {
    if(data.position + rect.width * rect.height >= data.length) {
        // catch buffer overflow
        return;
    }
    // copy data starting at data.position
    …
}
```

Adobe Flash CVE-2014-0556

- Integer overflow when `data.position` is almost 2^32
- The out-of-bounds write will be written before the start of the buffer because the pointer will wrap

```
bitmap = new BitmapData(1024, 1024);
rect = new Rectangle(0, 0, 1024, 1024);
array = new ByteArray();
array1.length = 0x40000000;
array1.position = 0xfffff000;
bitmap.copyPixelsToByteArray(rect, array1);
```

- Writeup: https://googleprojectzero.blogspot.com/2014/09/exploiting-cve-2014-0556-in-flash.html

A wild 0-day appears

- Someone posts an 0day to the TOR mailing list https://lists.torproject.org/pipermail/tor-talk/2016-November/042639.html
- Exploit source code analysis https://community.rapid7.com/community/metasploit/blog/2016/12/29/a-friendly-fireside-foray-into-a-firefox-fracas

Privileged Javascript Injection

```
1 | crmfObject = crypto.generateCRMFRequest("requestedDN", "regToken", "authenticator",
2 |                                         "escrowAuthorityCert", "CRMF Generation Done Code",
3 |                                         keySize1, "keyParams1", "keyGenAlg1",
4 |                                         ...,
5 |                                         keySizeN, "keyParamsN", "keyGenAlgN");
```

Argument	Description
"CRMF Generation Done Code"	This parameter is JavaScript to execute when the CRMF generation is complete.

Let's force a call to `generateCRMFRequest` from a privileged context

Privileged Javascript Injection

```
1  var y = {};
2  y.constructor.prototype.toString = function() {
3     crypto.generateCRMFRequest(..., run_payload, 1024, null, "rsa-ex");
4     return 5;
5  };
6  console.time(y);
```

`console.time` runs in a privileged browser context

CVE-2013-1710

```
1.1  --- a/security/manager/ssl/src/nsCrypto.cpp
1.2  +++ b/security/manager/ssl/src/nsCrypto.cpp
1.3  @@ -1916,17 +1916,17 @@ nsCrypto::GenerateCRMFRequest(nsIDOMCRMF
1.4     return NS_ERROR_FAILURE;
1.5    }
1.6    jsString = JS_ValueToString(cx, argv[4]);
1.7    NS_ENSURE_TRUE(jsString, NS_ERROR_OUT_OF_MEMORY);
1.8    argv[4] = STRING_TO_JSVAL(jsString);
1.9    JSAutoByteString jsCallback(cx, jsString);
1.10   NS_ENSURE_TRUE(!!jsCallback, NS_ERROR_OUT_OF_MEMORY);
1.11
1.12 - nrv = xpc->WrapNative(cx, ::JS_GetGlobalObject(cx),
1.13 + nrv = xpc->WrapNative(cx, JS_GetGlobalForScopeChain(cx),
1.14                  static_cast<nsIDOMCrypto *>(this),
1.15                  NS_GET_IID(nsIDOMCrypto), getter_AddRefs(holder));
1.16   NS_ENSURE_SUCCESS(nrv, nrv);
```

JS_GetGlobalForScopeChain() returns the global object for ***whatever function is currently running*** on the context.

CVE-2013-1670

```
2.12 +template <class Base>
2.13 +bool
2.14 +SecurityWrapper<Base>::defineProperty(JSContext *cx, HandleObject wrapper,
2.15 +                                       HandleId id, PropertyDescriptor *desc)
2.16 +{
2.17 +  if (desc->getter || desc->setter) {
2.18 +      JSString *str = IdToString(cx, id);
2.19 +      const jschar *prop = str ? str->getCharsZ(cx) : NULL;
2.20 +      JS_ReportErrorNumberUC(cx, js_GetErrorMessage, NULL,
2.21 +                             JSMSG_ACCESSOR_DEF_DENIED, prop);
2.22 +      return false;
2.23 +  }
2.24 +
2.25 +  return Base::defineProperty(cx, wrapper, id, desc);
2.26 +}
```

allows remote attackers to bypass certain read-only restrictions

Homework

Homework

- Fuzz a simple binary: fuzz_me.c

Tips

- **Don’t hesitate to ask questions! (Slack, email, etc)**

Homework grading

- Submit fuzzing results to cm7bv@virginia.edu with the subject “MST Assignment 5 - <YOUR_UVA_ID>”
  - eg: “MST Assignment 5 - cm7bv”
- Also, include a brief (1-paragraph) description of what you did and how it went

Useful Resources

- The Smart Fuzzer Revolution (https://www.youtube.com/watch?v=g1E2Ce5cBhI)
- More about crackaddr (http://2015.hackitoergosum.org/slides/HES2015-10-29%20Cracking%20Sendmail%20crackaddr.pdf)
- DARPA video about UVa's CGC team (https://www.youtube.com/watch?v=CHdmYY-kyuA)
- More about heap exploits (http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html)