

Security Considerations

Advanced Systems Programming (H/M)

Lecture 9

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Memory Safety
- Parsing and LangSec
- Modern Type Systems and Security

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Safety

- What is memory safety?
- Undefined behaviour in C and C++
- Security impact of memory un-safety
- Mitigations

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Safety in Programming Languages

- A **memory safe** language is one that ensures that the only memory accessed is that owned by the program, and that such access is done in a manner consistent of the declared type of the data
  - The program can access memory through its global and local variables, or through explicit references to them
  - The program can access heap memory it allocated via an explicit reference to that memory
  - All accesses obey the type rules of the language
- **Memory unsafe** languages fail to prevent accesses that break the type rules
  - C and C++ declare such behaviour to be **undefined**, but do nothing to prevent it from occurring

Memory Safe	Memory Unsafe
Java, Scala, C#, Rust, Go, Python, Ruby, Tcl, FORTRAN, COBOL, Modula-2, Occam, Erlang, Ada, Pascal, Haskell, …	C, C++, Objective-C

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (1/13)

- What types of memory unsafe behaviour can occur in C and C++?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (2/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation

```
double *d = malloc(sizeof(float));
```

A call to **`malloc()`** does not check if the amount of memory allocated corresponds to the size required to store the type

A memory safe language will require the size of the allocation to match the size of the allocated type at compile time

Operation ought to be “allocate enough memory for an object of type **`T`** and return a reference-to-**`T`**” and not “allocate *n* bytes of memory and return an untyped reference”

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (3/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation

```
char *buffer;

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    perror("Cannot receive");
} else {
    // Handle data
}
```

Passes a pointer to the **`buffer`** to the **`recv()`** function, but forgets to **`malloc()`** the memory – or assumes it'll be allocated in **`recv()`**

Memory safe languages require that all references be initialised and refer to valid data – good C compilers warn about this too

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (4/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  char *x = malloc(14);
  sprintf(x, "Hello, world!");
  free(x);
  printf("%s\n", x);
}
```

Accesses memory that has been explicitly freed, and hence is no longer accessible

Automatic memory management eliminates this class of bug

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (5/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free

```
int *foo() {
  int n = 42;
  return &n;
}
```

Return reference to stack allocated memory that is used after the stack frame has been destroyed

Automatic memory management eliminates this class of bug

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (6/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type

```
char *buffer = malloc(BUFLEN);

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    …
} else {
    struct pkt *p = (struct pkt *) buffer;
    if (p->version != 1) {
        …
    }
    …
}
```

Common in C code to see casts from **`char *`** buffers to more specific types (e.g., to a **`struct`** representing a network packet format)

Efficient – no memory copies – but unsafe since makes assumptions of **`struct`** layout in memory, size of block being cast

Memory safe language disallow arbitrary casts – write conversion functions instead; eliminates undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (7/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values

```
// Send the requested file
while ((rlen = read(inf, buf, BUFLEN)) > 0) {
  if (send_response(fd, buf, strlen(buf)) == -1) {
    return -1;
  }
}
```

Strings in C are zero terminated, but **`read()`** does not add a terminator; buffer overflow results

Memory safe languages apply string bounds checks – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (8/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow

```
#define BUFSIZE 256
int main(int argc, char *argv[]) {
    char *buf;
    buf = malloc(sizeof(char) * BUFSIZE);
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply bounds checks to heap allocated memory – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (9/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow

```
int main(int argc, char *argv[]) {
    char buf[256];
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply array bounds checks – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (10/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic

`sizeof()` returns size in bytes, arithmetic on **`int *`** pointers is on pointer sized values; bounds check is too lax

Memory safe languages disallow arbitrary pointer arithmetic

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && (buf_ptr < (buf + sizeof(buf)))) {
  *buf_ptr++ = parseint(getdata());
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (11/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic
  - Use of uninitialised memory

Memory allocated with **`malloc()`** has undefined contents

Memory safe languages require memory to be initialised, or mandate that the runtime fills with known value

```
static char *
read_headers(int fd)
{
  char     buf[BUFLEN];
  char    *headers   = malloc(1);
  size_t   headerLen = 0;
  ssize_t  rlen;

  while (strstr(headers, "\r\n\r\n") == NULL) {
    rlen = recv(fd, buf, BUFLEN, 0);
    if (rlen ==  0) {
      // Connection closed by client
      free(headers);
      return NULL;
    } else if (rlen < 0)  {
      free(headers);
      perror("Cannot read HTTP request");
      return NULL;
    } else {
      headerLen += (size_t) rlen;
      headers = realloc(headers, headerLen + 1);
      strncat(headers, buf, (size_t) rlen);
    }
  }
  return headers;
}
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (12/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic
  - Use of uninitialised memory
  - Use of memory via dangling pointer
  - Use of memory via **`null`** pointer

`lookup()` call may fail, returning **`null`** pointer

Memory safe languages either: fail safely with an exception; or use **`Option<>`** types to enforce that the null pointer check is done

```
struct user *u = lookup(db, key);
printf("%s\n", u->name);
```

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Undefined Behaviour (13/13)

- What types of memory unsafe behaviour can occur in C and C++?
  - …and more https://cwe.mitre.org/data/definitions/658.html

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Impact of Memory Unsafe Languages

- Lack of memory safety breaks the machine abstraction
  - With luck, program crashes – segmentation violation
  - If unlucky, memory unsafe behaviour corrupts other data owned by program
    - Undefined behaviour occurs
    - Cannot predict without knowing precise layout of program in memory
    - Difficult to debug
    - **Potential security risk** – corrupt program state to force arbitrary code execution

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Impact of Memory Unsafe Languages – Security (1/2)

Vulnerabilities By Type

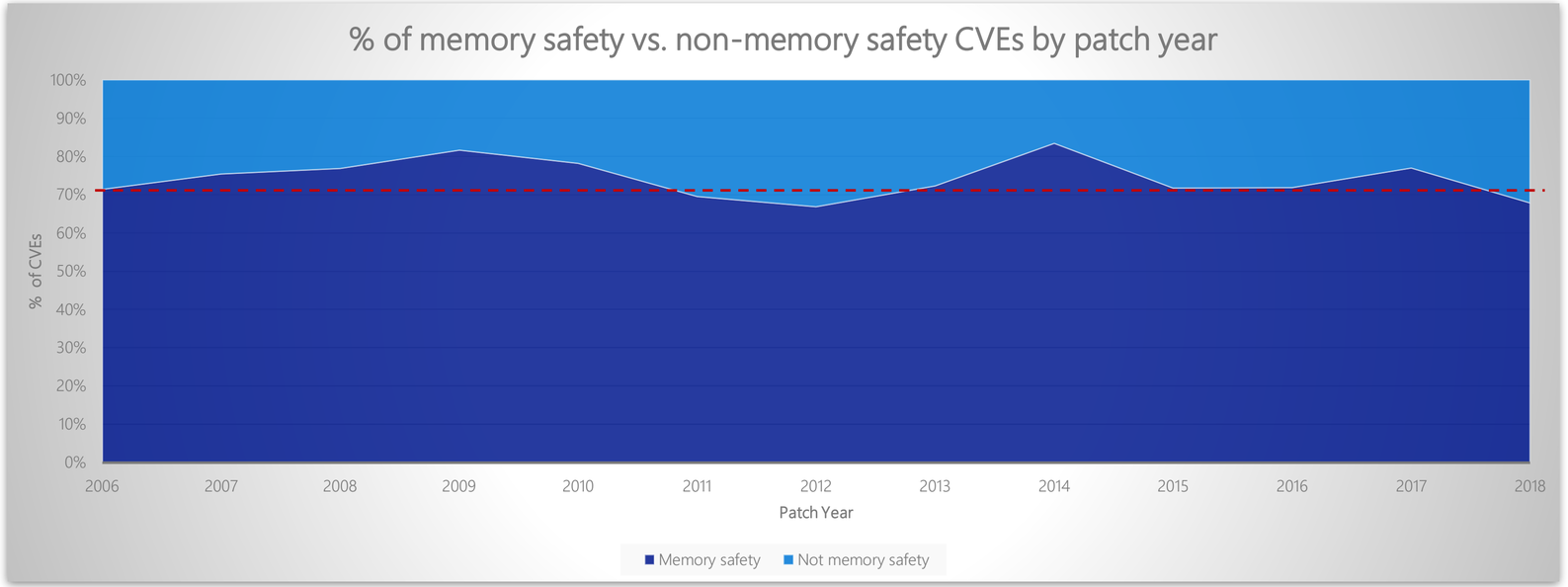
Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	894	177	112	172			2	7		25	16	103			2
2000	1020	257	208	206		2	4	20		48	19	139			
2001	1677	403	403	297		7	34	123		83	36	220		2	2
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1527	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	111	12	145	96	134	5	38	5
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	14
2006	6610	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	6520	1101	2601	954	95	706	884	339	14	267	324	242	69	700	44
2008	5632	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322	9	337	302	223	115	138	738
2010	4652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	5297	1425	1459	843	423	243	758	122	13	343	389	250	166	14	624
2013	5191	1455	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	7946	1598	1574	850	420	305	1105	204	12	457	2104	239	264	2	401
2015	6484	1791	1826	1079	749	218	778	150	12	577	748	367	248	5	127
2016	6447	2028	1494	1325	717	94	497	99	15	444	843	600	87	7	1
2017	14714	3154	3004	2805	745	503	1516	274	11	629	1706	459	327	18	6
2018	16555	1852	3035	2451	400	516	2001	509	11	709	1374	247	461	31	4
2019	4	2				1									
Total	110603	22685	30435	17226	5043	7438	13667	3823	162	5880	10104	4877	2123	2195	4333
% Of All		20.5	27.5	15.6	4.6	6.7	12.4	3.5	0.1	5.3	9.1	4.4	1.9	2.0	

Source: https://www.cvedetails.com/vulnerabilities-by-types.php

- Half of **all** security vulnerabilities are memory safety violations that should be caught by a modern type system
  - Buffer overflows
  - Memory corruption
  - Treating data as executable code
- Use of type-based modelling of the problem domain can help address others – by more rigorous checking of assumptions

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Impact of Memory Unsafe Languages – Security (2/2)



Source: Matt Miller, Microsoft, presentation at BlueHat IL conference, February 2019
https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

- ~70% of Microsoft security updates fix bugs relating to unsafe memory usage
- This has not significantly changed in >10 years
- We're **not** getting better at writing secure code in memory unsafe languages

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mitigations for Memory Unsafe Languages (1/2)

- Use modern tooling for C and C++ development:
  - Compile C code with, at least, **`clang –W -Wall -Werror`**
    - Review documentation to find additional warnings it makes sense to enable
    - Fix **all** warnings – let the compiler help you debug your code
  - Use **`clang`** static analysis tools during debugging:
    - https://clang.llvm.org/docs/AddressSanitizer.html
    - https://clang.llvm.org/docs/MemorySanitizer.html
    - https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
    - https://clang.llvm.org/docs/ThreadSafetyAnalysis.html
    - https://clang.llvm.org/docs/ThreadSanitizer.html
    - These have very high overhead, but catch many memory and thread safety problems

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Mitigations for Memory Unsafe Languages (2/2)

- Use modern C++ language features:
  - I advise against C++ programming – language is too complex to understand
    - Too many features have been added over time, too few removed; code mixes old and new features
    - Very few people know the whole language
    - In hindsight, some C++ defaults were inappropriate
    - Modern C++ addresses many of these issues – but must retain backwards compatibility
      - Rust adopts many lessons learned in the development of C++
      - And can be simpler, since doesn't need worry about compatibility with legacy code
  - But if you have to use C++, modernise the code base, to use newer – safer – language features and idioms, where possible

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Gradually Rewrite code into a Memory Safe Language

- Consider rewriting the most critical sections of the code in a memory safe language:
  - Gradually rewrite C into Rust
    - Rust can call C functions directly
    - Rust can be compiled into a library that can be directly linked with C or C++
      - https://doc.rust-lang.org/nomicon/ffi.html
      - https://unhandledexpression.com/rust/2017/07/10/why-you-should-actually-rewrite-it-in-rust.html
    - Possible to do a gradual rewrite of C or C++ into a safe language, while keeping the application usable and building at each stage
  - Gradually rewrite Objective-C into Swift
    - In the Apple ecosystem, similarly possible to gradually rewrite applications into Swift
- **Difficult** and requires languages with compatible runtime models
  - Gradual rewrite likely has more chance of success than from-scratch rewrite

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Safety

- What is memory safety?
- Undefined behaviour in C and C++
- Security impact of memory un-safety
- Mitigations

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow