



DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

Bevezetés a memória korrupciós hibákba

VIHIBB01 – Kódolás és IT biztonság (2022)

Gazdag András

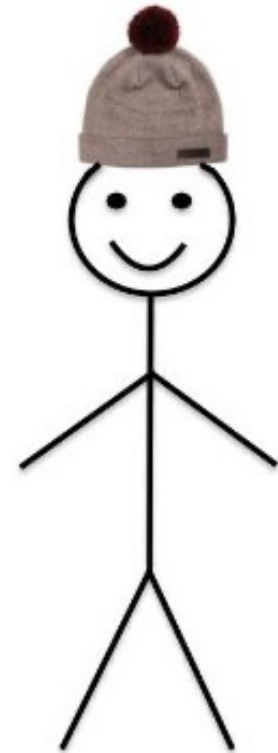
CrySyS Lab, BME

andras.gazdag@crysys.hu

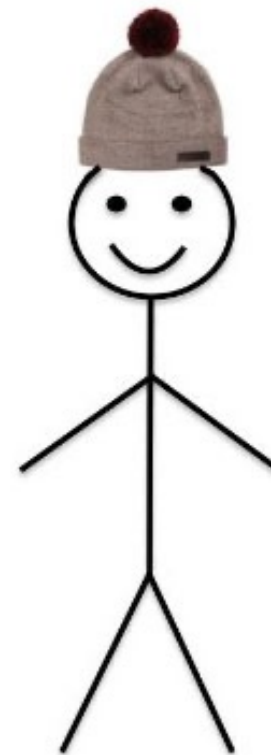


M Ű E G Y E T E M 1 7 8 2

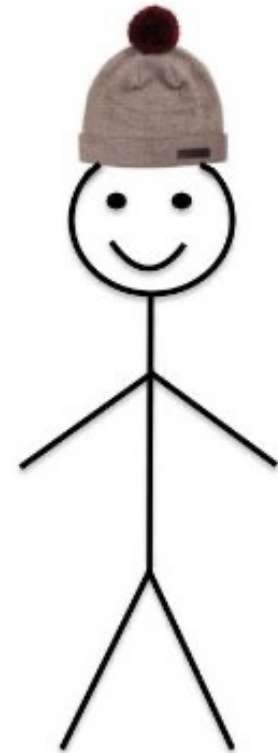
- Ő itt Robert.
- Robert egy teenager a '80-as években.
- Robert hall egy fantasztikus új dologról: az Internet!
- Robertnek eszébe jut egy érdekes kérdés...
- Mekkora is az Internet?



- Robert ír egy szoftvert...
- Amivel gépről gépre tud ugrani és számolni.
- Felhasználói beavatkozás nélkül.
- Ma egy ilyen úgy hívunk, hogy...

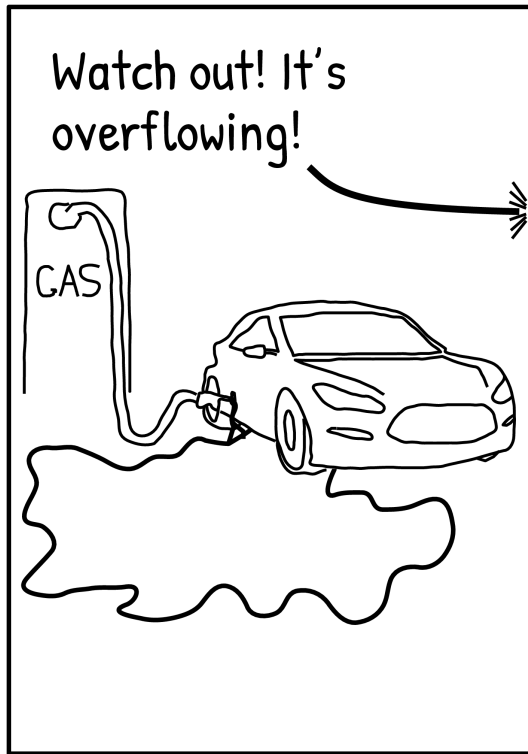


- Robert szabadon engedi a szoftverét.
- Robert ezzel 10-100 millió dollár közötti kárt okoz.
- Robert ezután börtönbe kerül... majdnem.

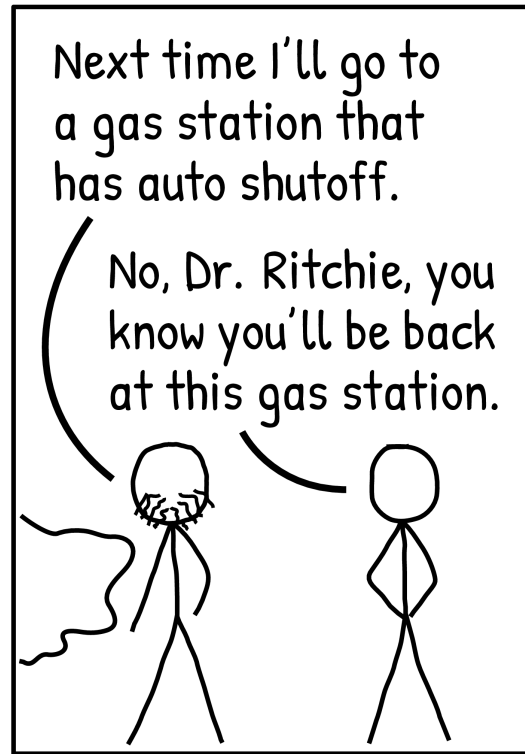


- Ő itt Robert Morris.
- Roberts apja az NSA egyik vezető tudósa
- Robert írta az első vírust, ami kihasználta egy *buffer overflowt*.
- Robert ma is egy professzor az MIT-n.





Buffer Overflow.



icanbarelydraw.com CC BY-NC-ND 3.0

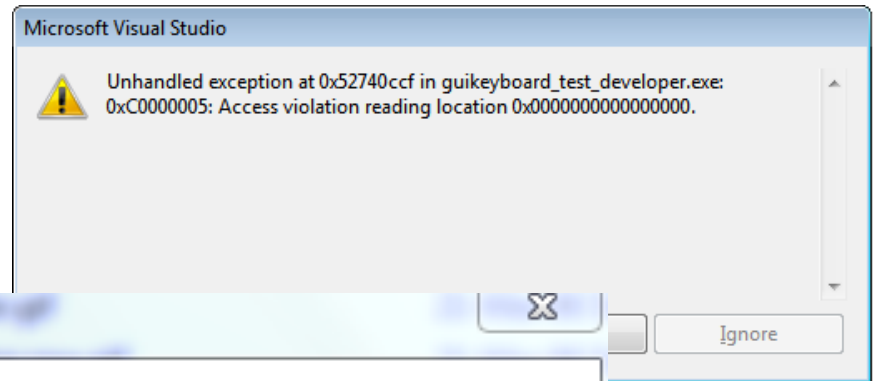
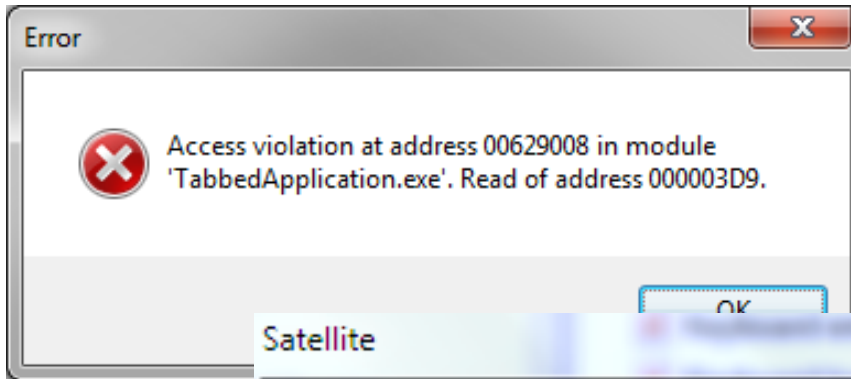


MEMÓRIA KORRUPCIÓ

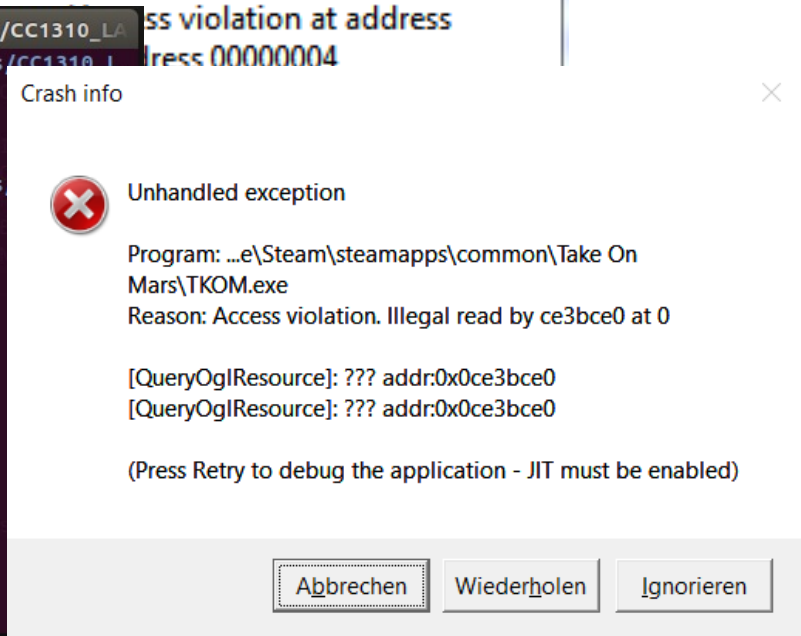
Memória korrupció

- Akkor történik, ha a memória egy része véletlenül vagy szándékosan módosul egy programozási hiba miatt
 - Támadások sem mindig tudják vagy akarják célzottan módosítani a memóriát
 - A módosulás maga is elég lehet a támadónak
- Ha a módosított memória tartalmát a program később felhasználja, akkor ez összeomláshoz vagy véletlen viselkedéshez vezethet
- Példák memória korrupcióra:
 - Buffer overflow (puffer túlcsordulás) a stacken vagy a heapen
 - Integer overflow (integer túlcsordulás)
 - NULL pointer hibák vagy dangling (lógó) pointer hibák
- Memória korrupcióhoz vezető hibákat tipikusan nehéz megtalálni egy forráskódban (nagyrészt emiatt marad sok ilyen hiba a kész termékekben)

Memória korrupció



```
tomer@ubuntu: ~/ti/simplelink_cc13x0_sdk_1_30_00_06/kernel/tirtos/builds/CC1310_LA...
tomer@ubuntu:~/ti/simplelink_cc13x0_sdk_1_30_00_06/kernel/tirtos/builds/CC1310_LA...
AUNCHXL/debug/gcc$ make
Segmentation fault (core dumped)
makefile:39: recipe for target 'all' failed
make: *** [all] Error 139
tomer@ubuntu:~/ti/simplelink_cc13x0_sdk_1_30_00_06/kernel/tirtos/builds/CC1310_LA...
AUNCHXL/debug/gcc$
```



Buffer overflow

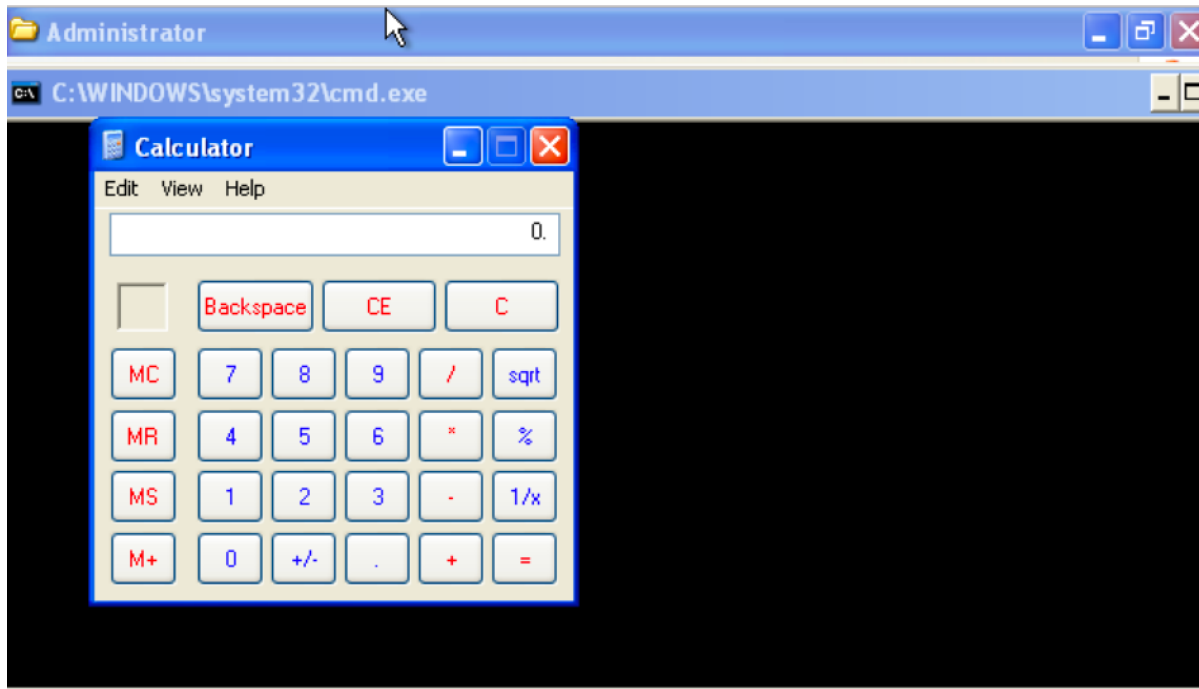
- Akkor történik, ha egy buffer tartalma túllépi a számára foglalt memória területet
- Általában a program ilyenkor összeomlik
 - Szegmentációs hibával Linux környezetben
 - Access violation hibával Windowson
- Támadó kihasználhatja a sérülékenységet úgy, hogy értékes területeket (visszatérési cím, pointerok) átír egy támadás során
- Kényszeríteni lehet vele egy programot arra, hogy változzon a végrehajtási útvonala így
 - Legrosszabb esetben a memóriába bejuttatott tartalmat kezdheti futtatni

Stack overflow

- Speciális esete a buffer overflownak
- Akkor fordulhat elő, ha egy függvény egy kívülről érkező tartalmat egy lokális bufferébe másolja méret ellenőrzés nélkül
- Így egy kívülről érkező adat felülírhat más értékeket a stacken, akár a visszatérési címet is
 - Amikor a függvény végrehajtás végén a program visszatérne a függvényből, nem oda fog ugrani, ahova kellene
 - Így lehetséges átvenni az irányítást a végrehajtási útvonal felett
- Ha a módosított cím egy támadó által kontrollált tartalomra mutat, akkor az ilyenkor végrehajtásra kerül
 - Így lehetséges módosítani a végrehajtandó kódot

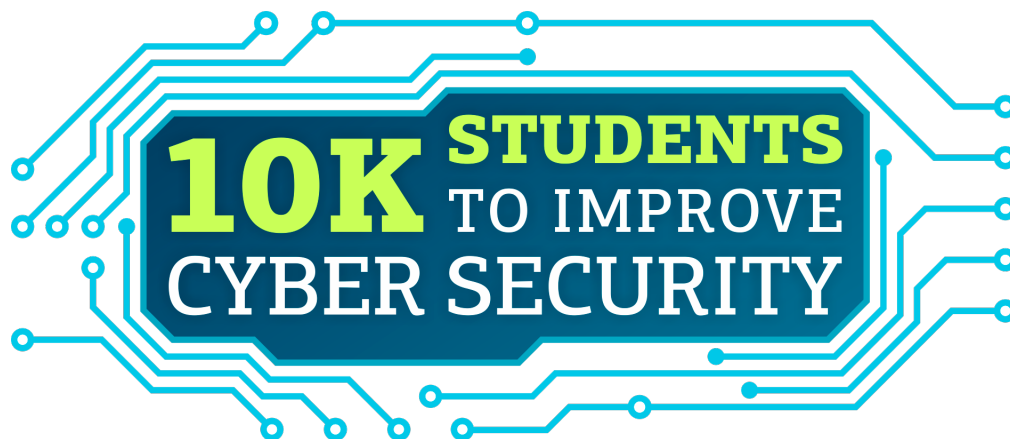
Nagyobb overflow támadások

- Morris worm (1988): overflow a fingerd programban
 - 6000 fertőződött meg (10% az Internetnek)
- CodeRed (2001): overflow a Microsoft IIS szerverben
 - 300 000 fertőződött meg 14 óra alatt
- SQL Slammer (2003): overflow a MS-SQL szerverben
 - 75 000 gép fertőződött meg (90%-a az összes sérülékeny gépnek az interneten) 10 perc alatt
- 2006 és 2008 között az újonnan talált sérülékenységek több mint fele buffer overflow hiba volt
- Manapság a webes támadások száma nagyobb, ugyanakkor a buffer overflow alapú támadások még mindig széles körben kerülnek kihasználásra
 - Az IoT és az Industry 4.0 terjedése sokat ront ezen a helyzeten



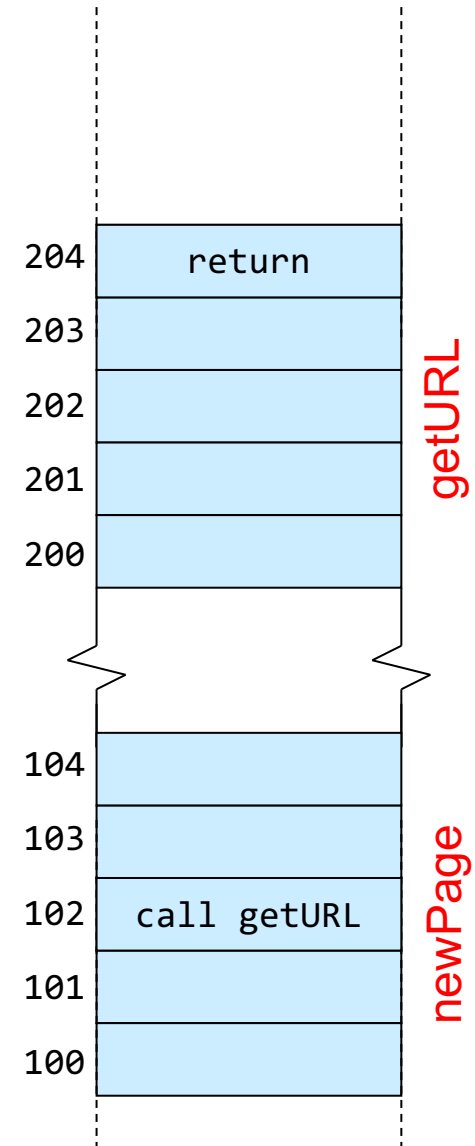
A STACK OVERFLOW TÁMADÁS

Az itt bemutatott diák részben Prof. Herbert Bos (VU Amsterdam) munkáján alapulnak, amit a SysSec Network of Excellence keretében publikált a 10K students to improve cyber security kezdeményezés részeként.



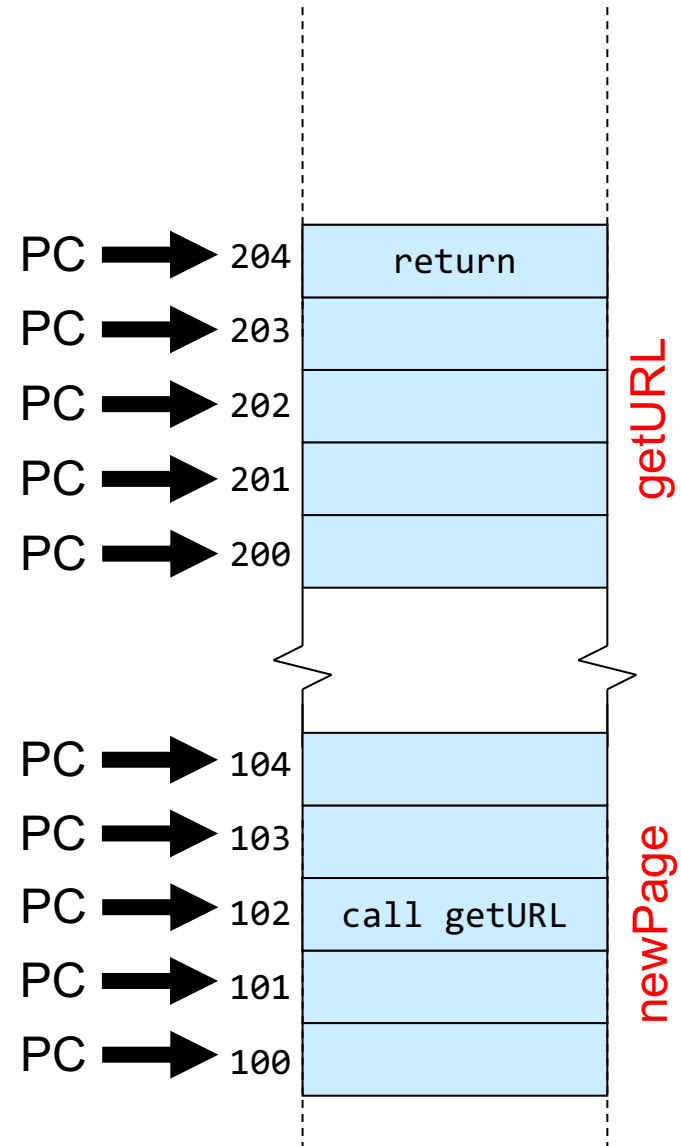
Egy program végrehajtása

- Utasítások sorozata a memóriában
- Logikailag függvényekre van osztva, amelyek egymást hívják meg valamilyen sorrendben
 - Pl: a `newPage()` függvény meghívhatja a `getURL()` függvényt



Egy program végrehajtása

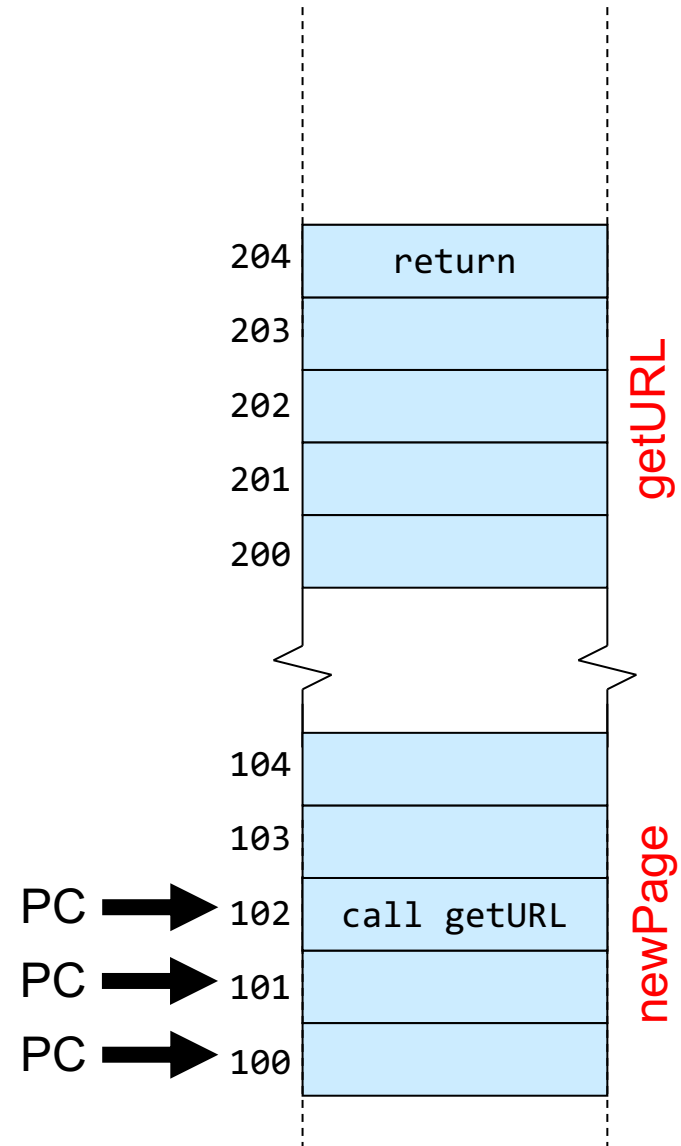
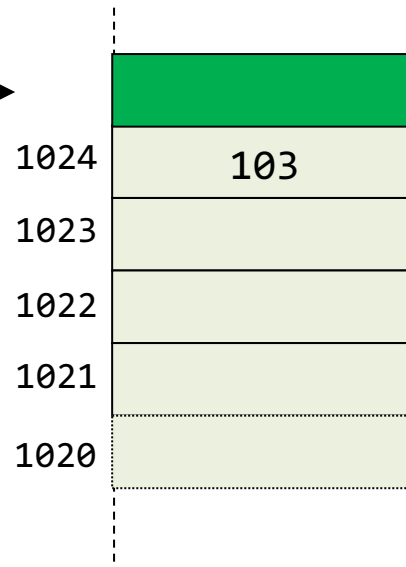
- Utasítások sorozata a memóriában
- Logikailag függvényekre van osztva, amelyek egymást hívják meg valamilyen sorrendben
 - Pl: a `newPage()` függvény meghívhatja a `getURL()` függvényt
- A processzorban a program counter (PC) tárolja a következőnek végrehajtandó utasítás címét
 - Átlagos esetben a PC értéke egyesével kerül növelésre
 - » A 100. címen lévő utasítást a 101. címen lévő utasítás végrehajtása követi
 - ***Kivéve, ha egy függvényhívás történik***



Egy program végrehajtása

- Honnan tudja a CPU, hogy hova kell visszatérnie?
 - Eltárolja a visszatérési címet a stacken

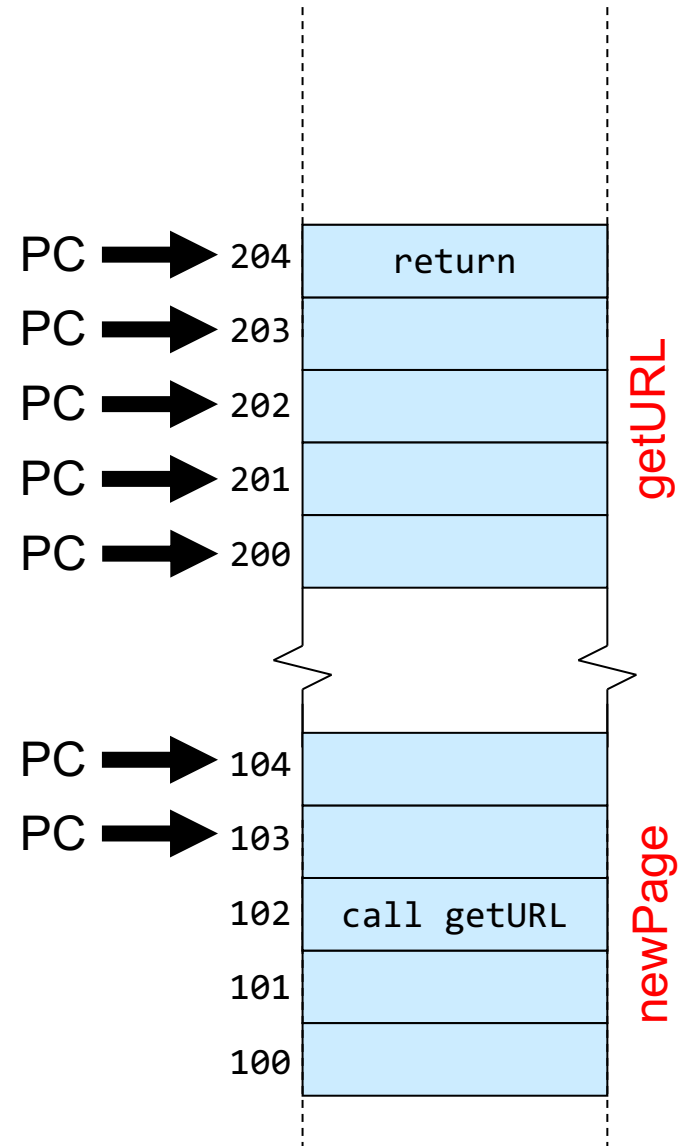
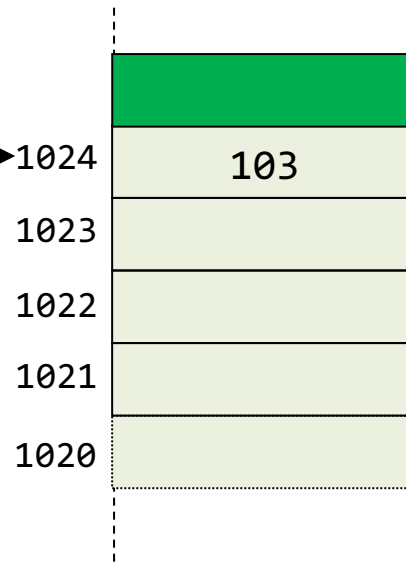
stack pointer (SP) →
Az utolsó értékre mutat a stacken;
A stack ellentétes irányba nő, mint amerre a végrehajtás halad!



Egy program végrehajtása

- Honnan tudja a CPU, hogy hova kell visszatérnie?
 - Eltárolja a visszatérési címet a stacken

stack pointer (SP) →
Az utolsó értékre mutat a stacken;
A stack ellentétes irányba nő, mint amerre a végrehajtás halad!



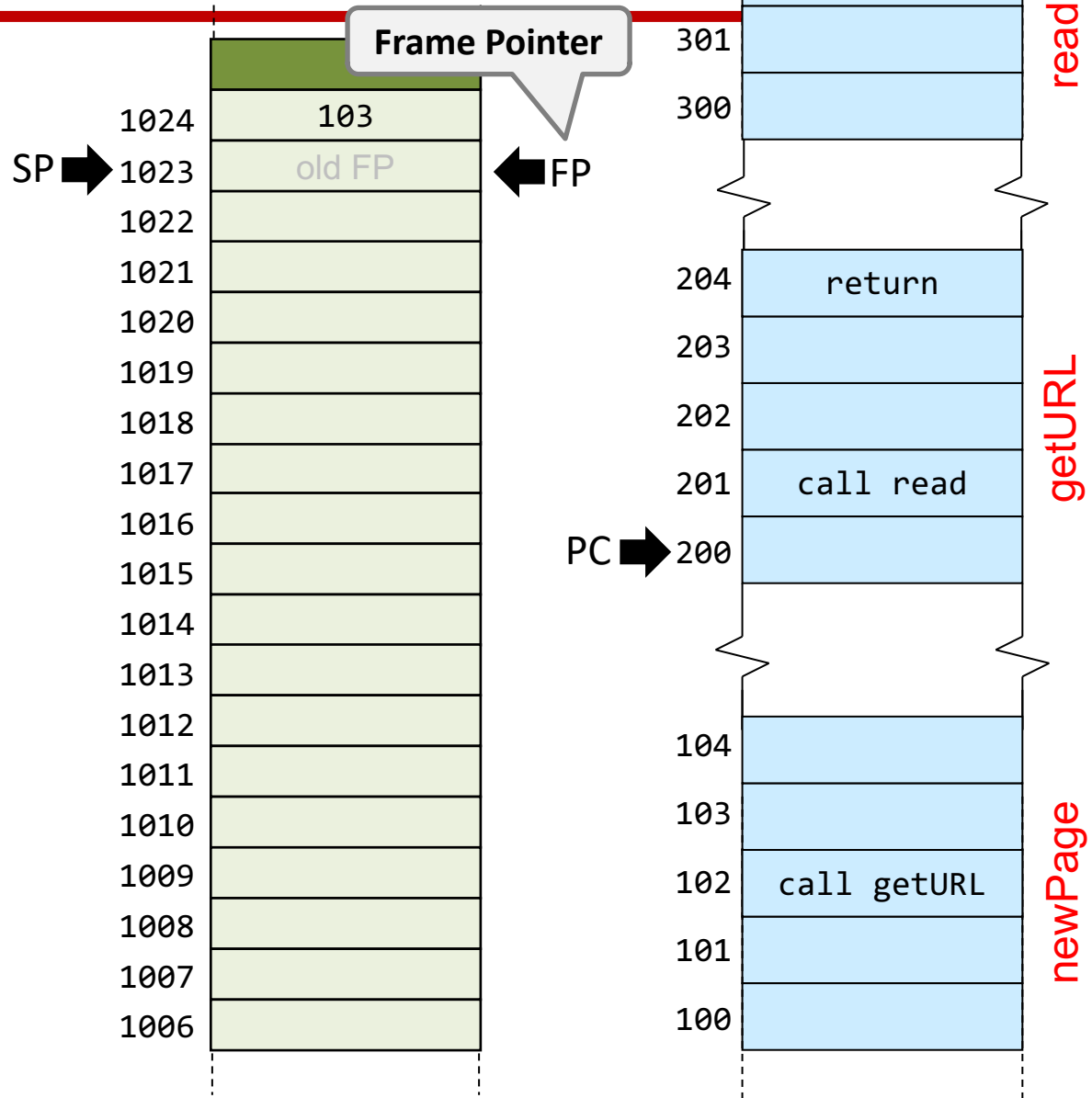
Függvény változókkal

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}
```

```
newPage()  
{  
    getUrl();  
}
```

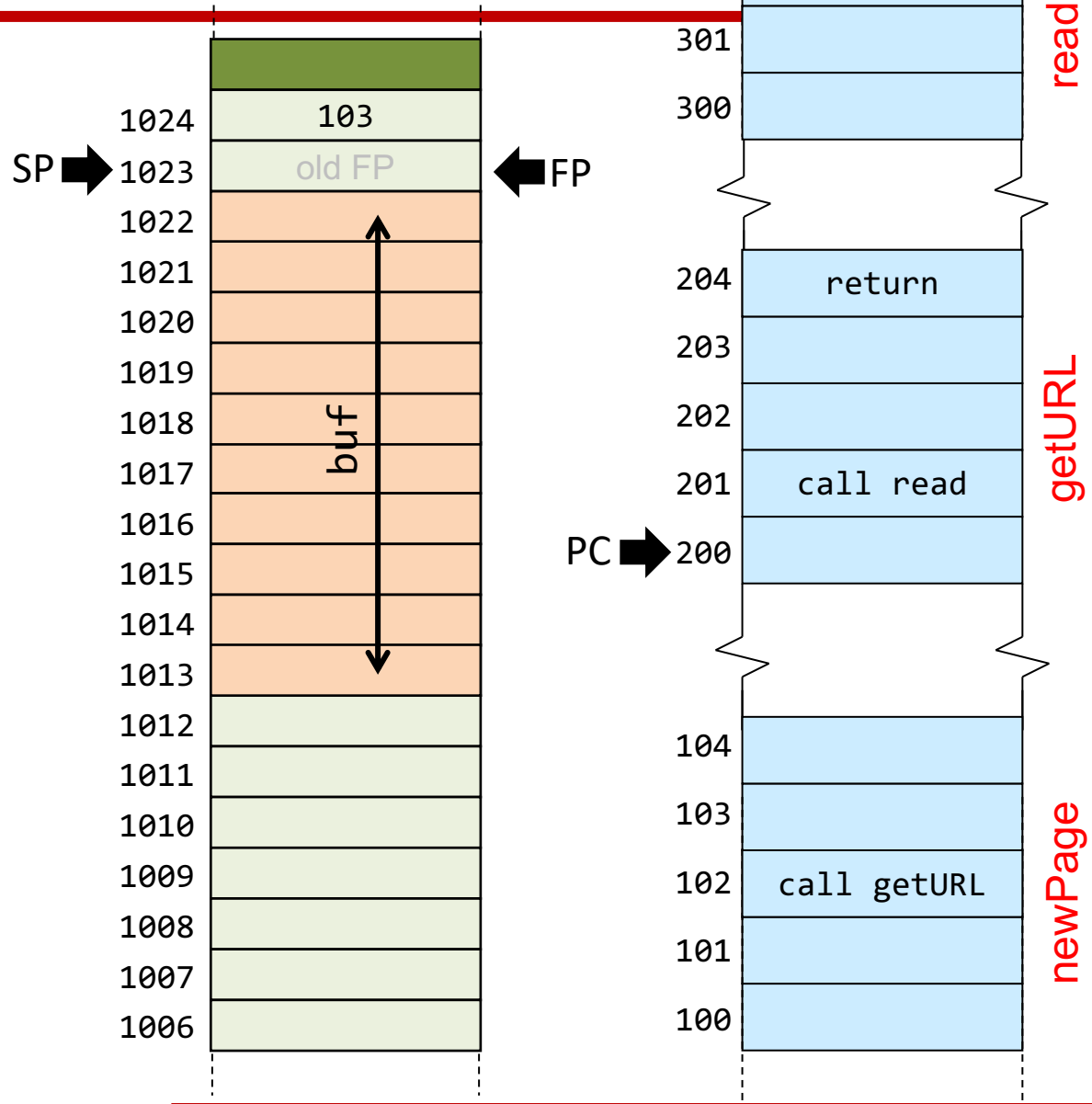
Amikor getUrl() meghívódik...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



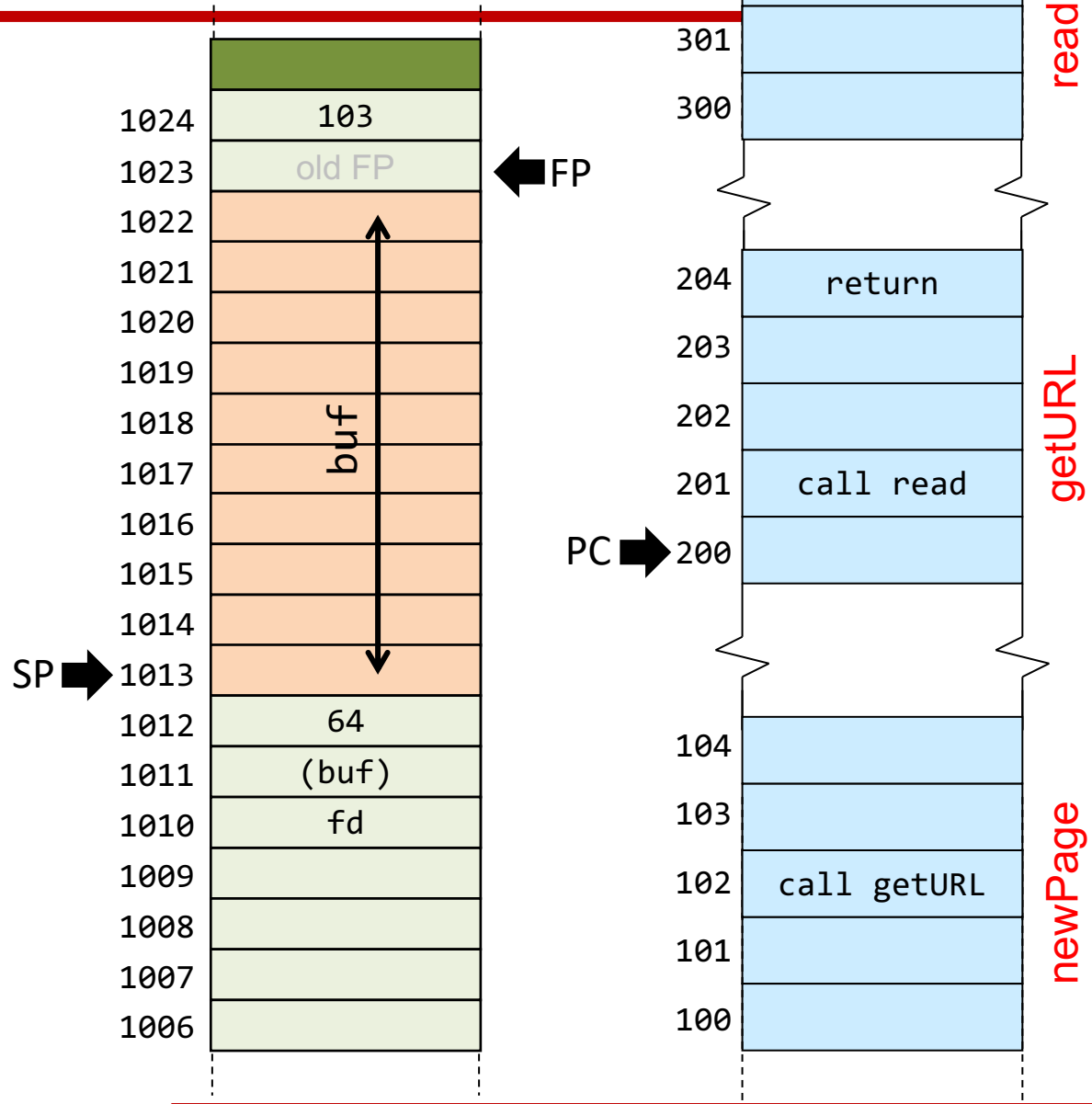
Amikor getUrl() meghívódik...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



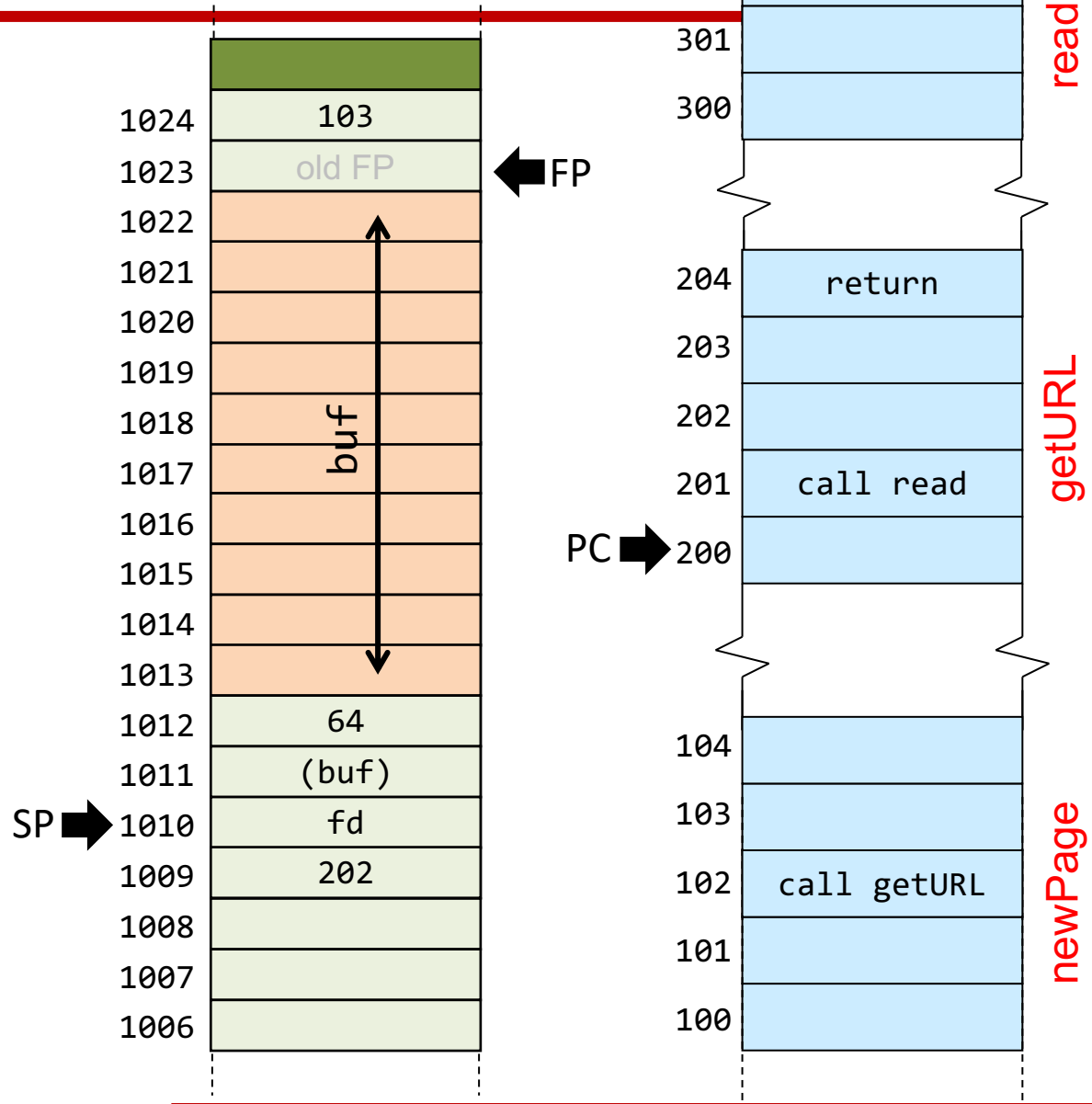
Amikor getUrl() meghívódik...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



Amikor getUrl() meghívódik...

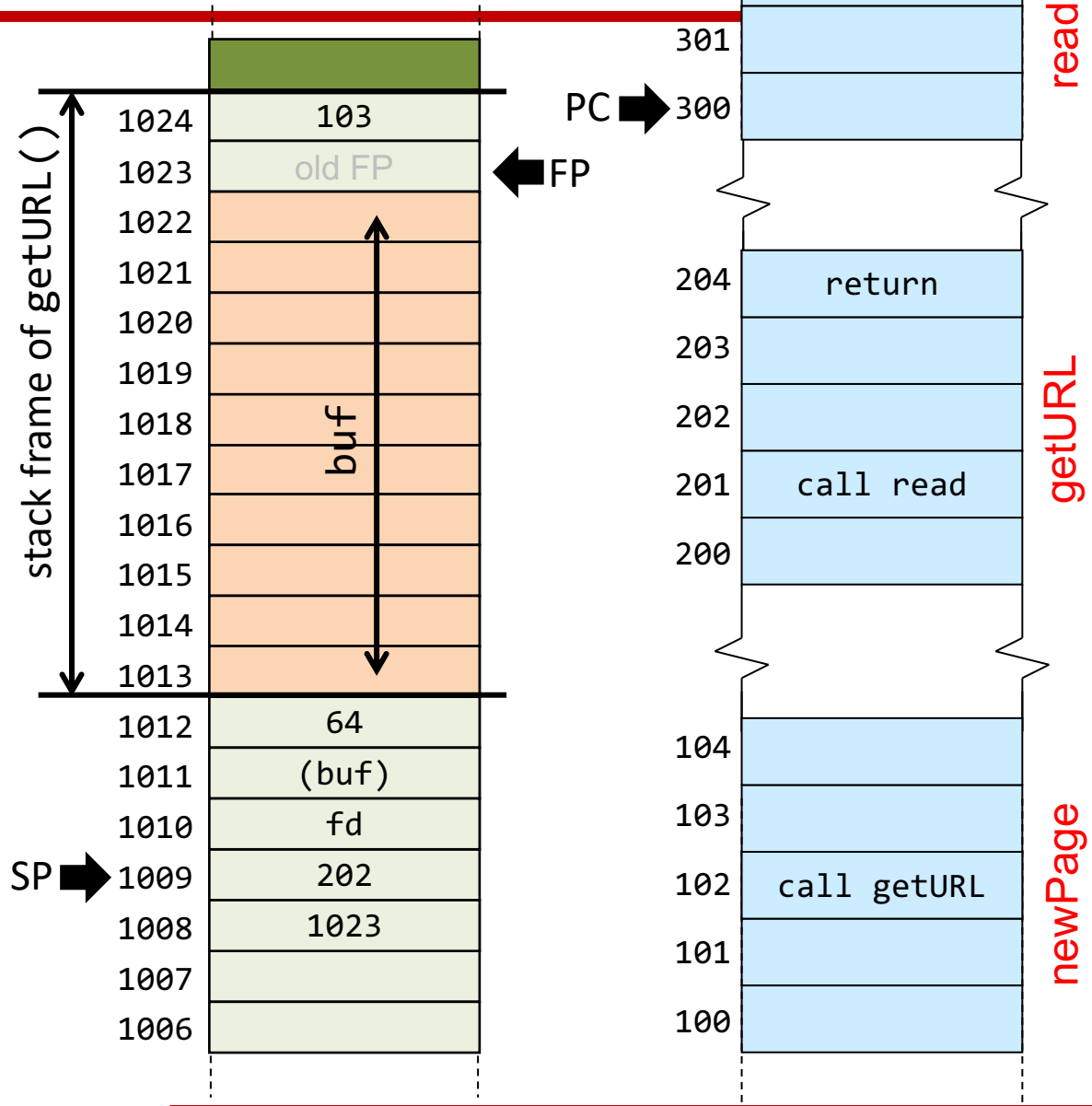
```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



Amikor getUrl() meghívódik...

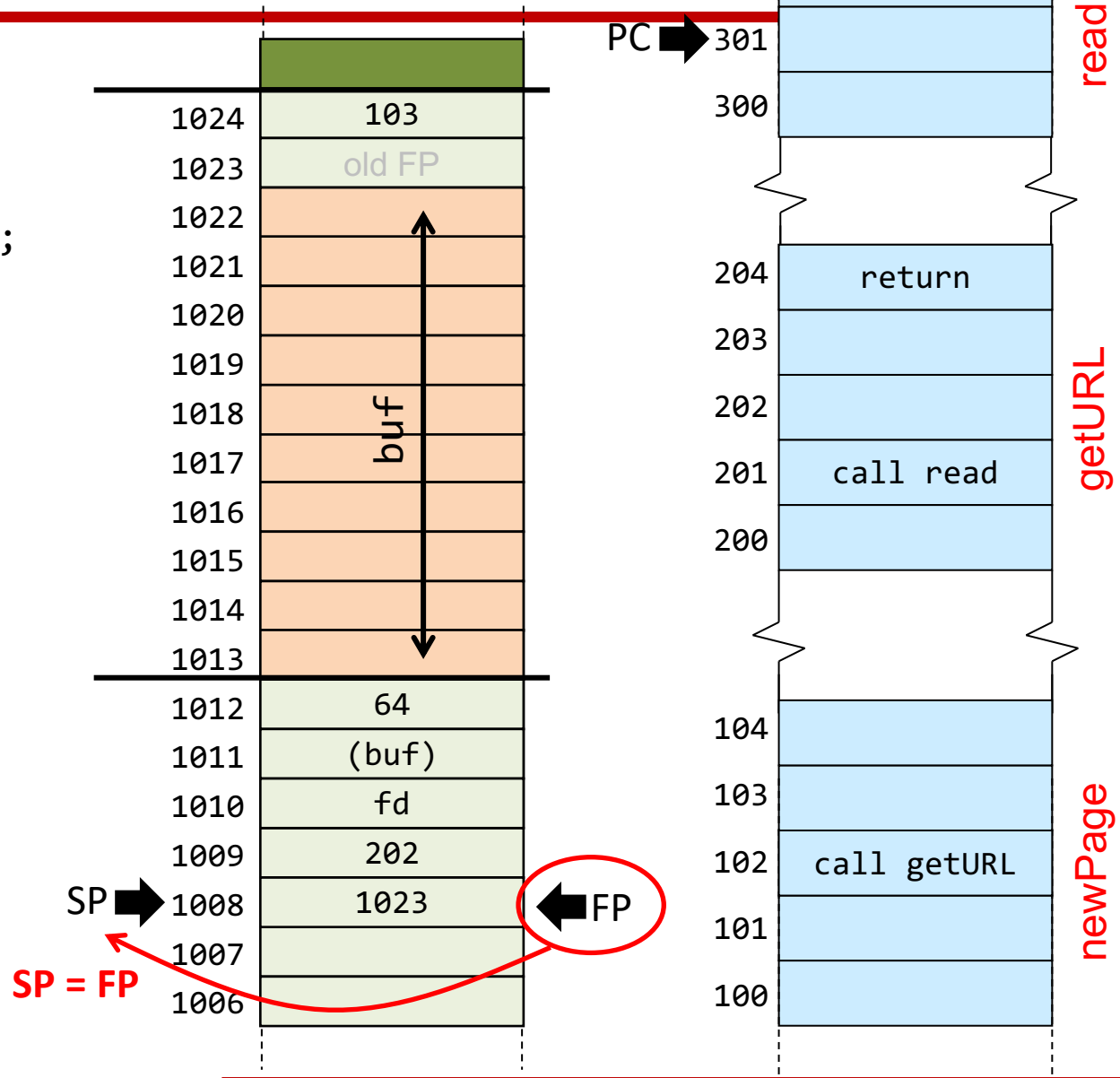
```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```

push FP
FP = SP



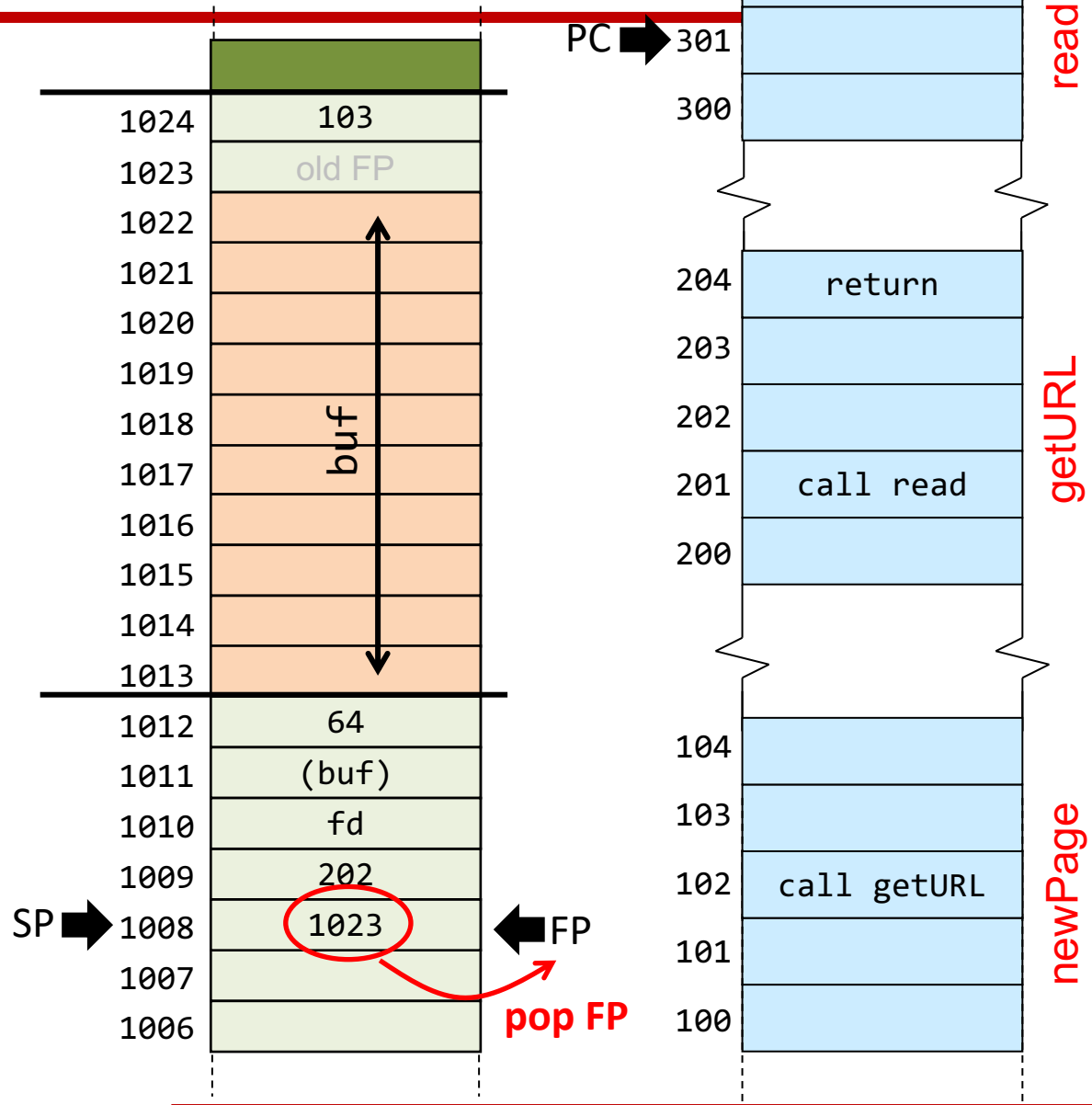
Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



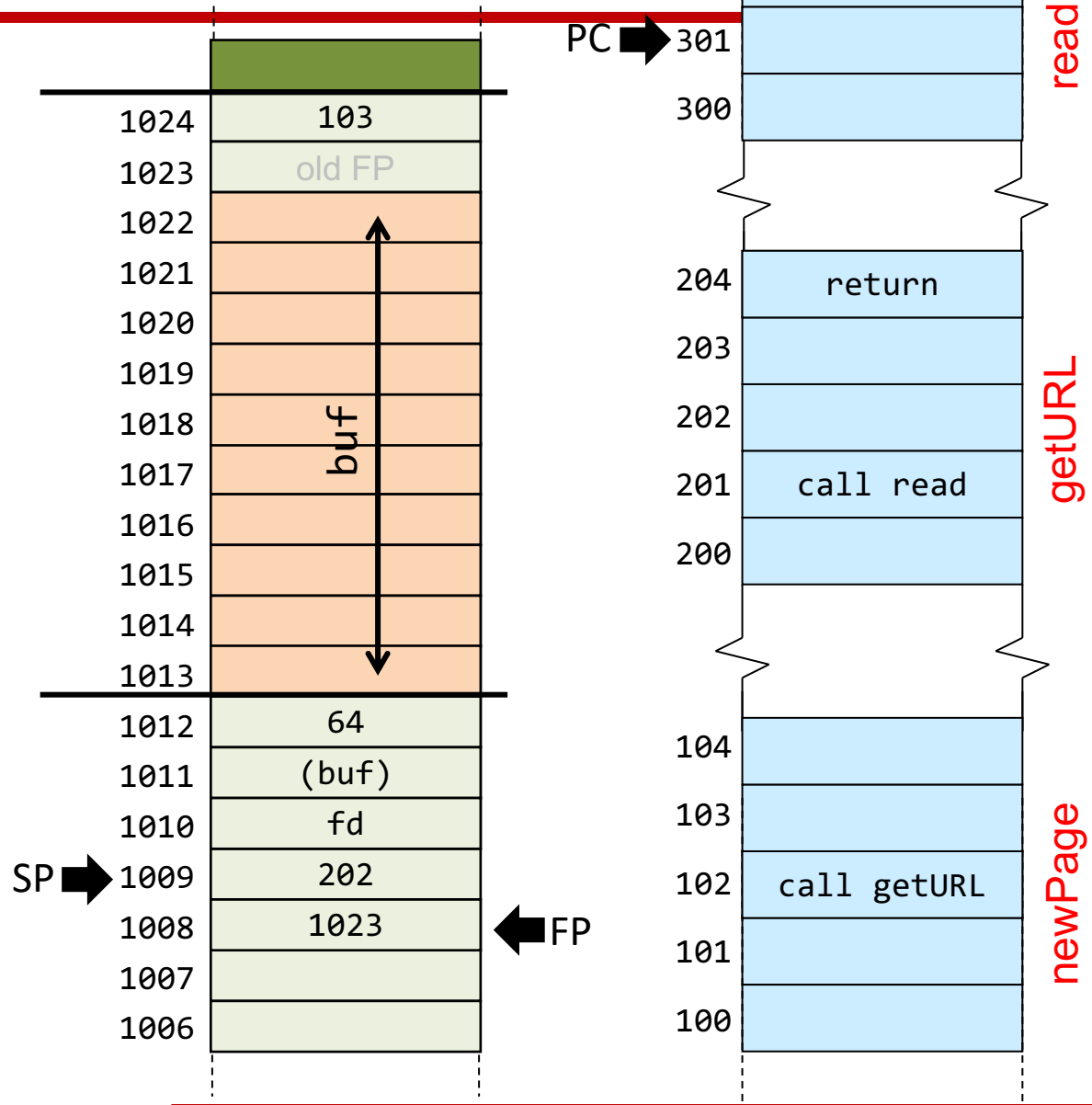
Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



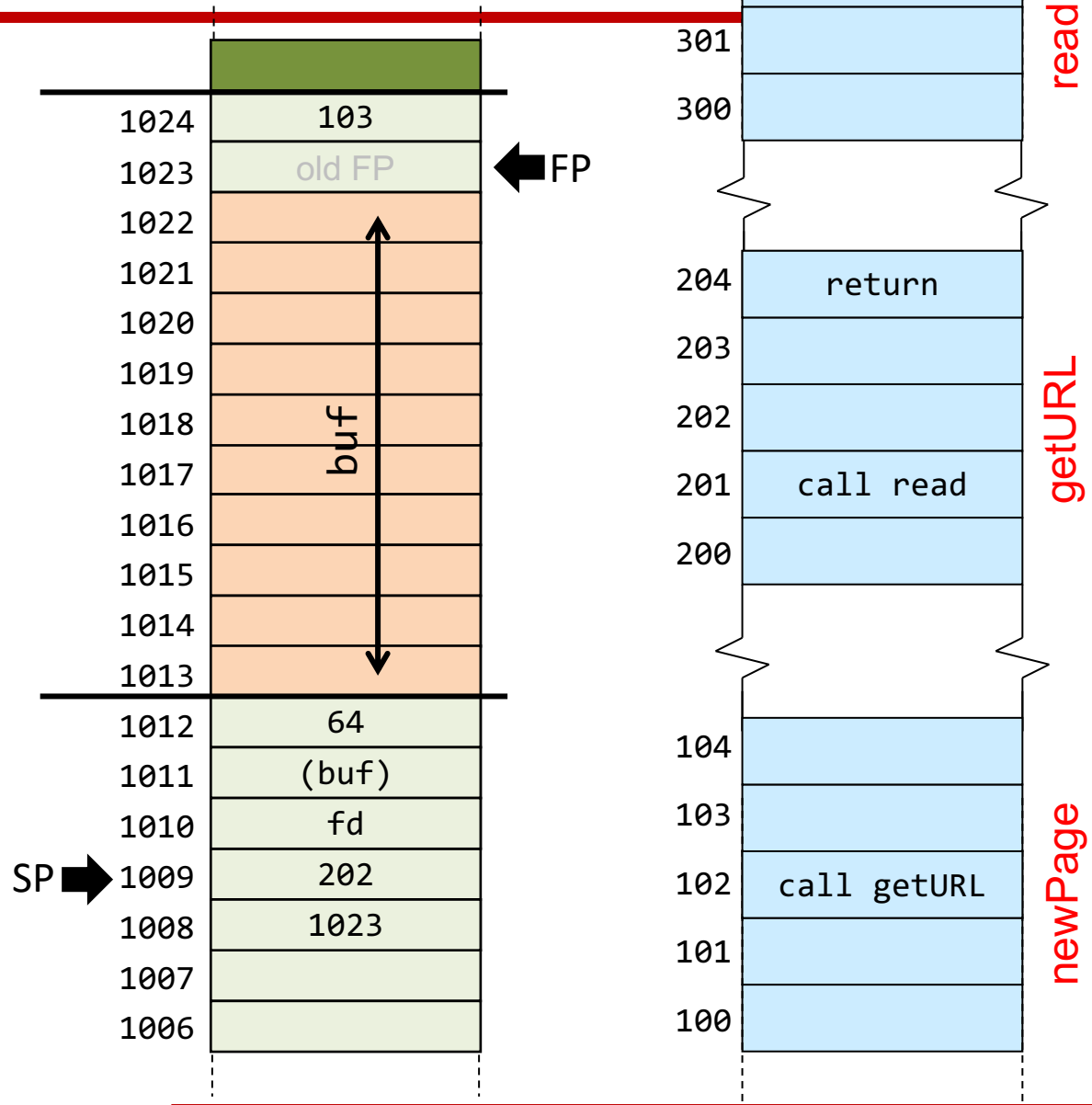
Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



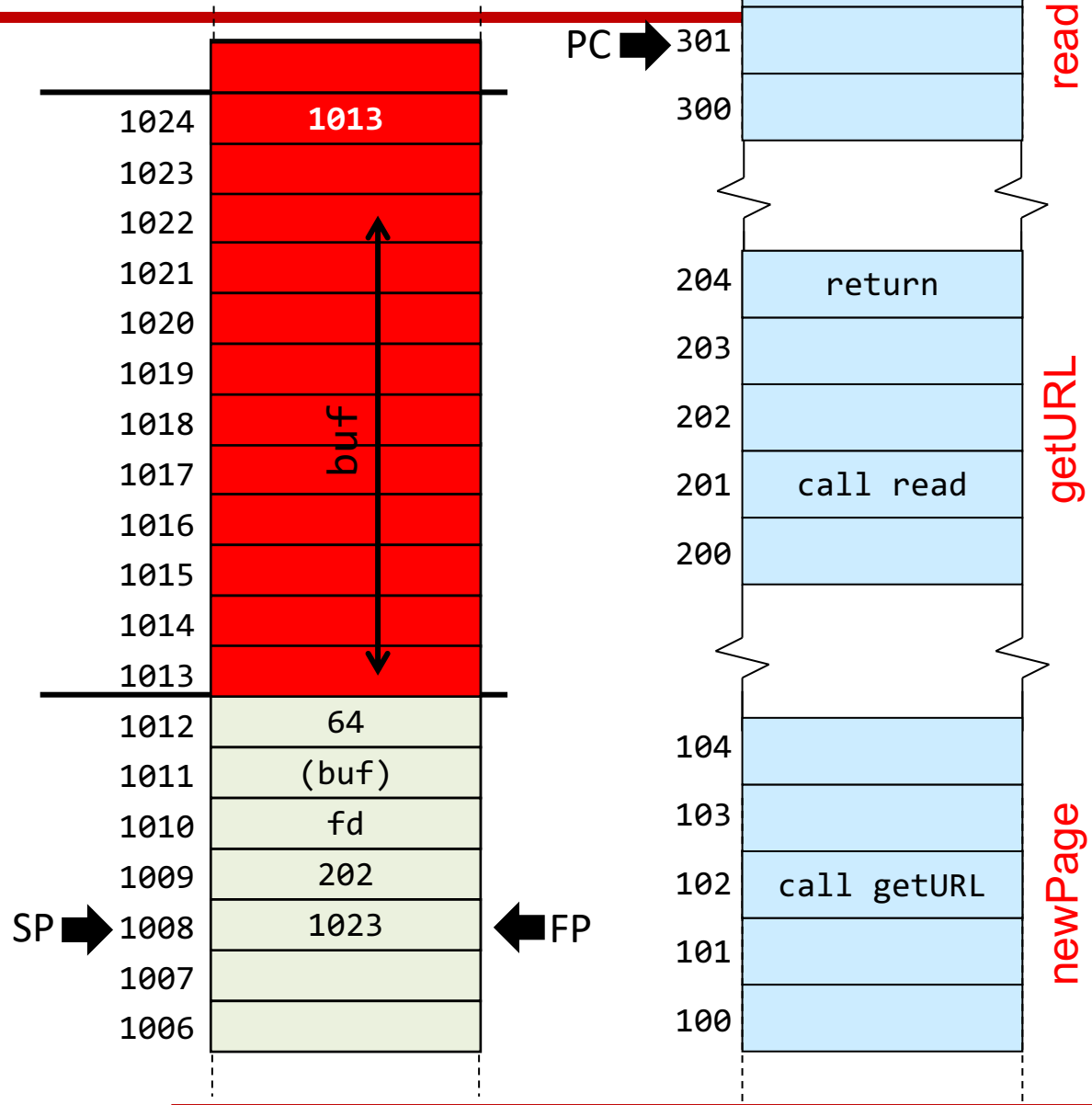
Mi a probléma?

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}
```

```
newPage()  
{  
    getUrl();  
}
```

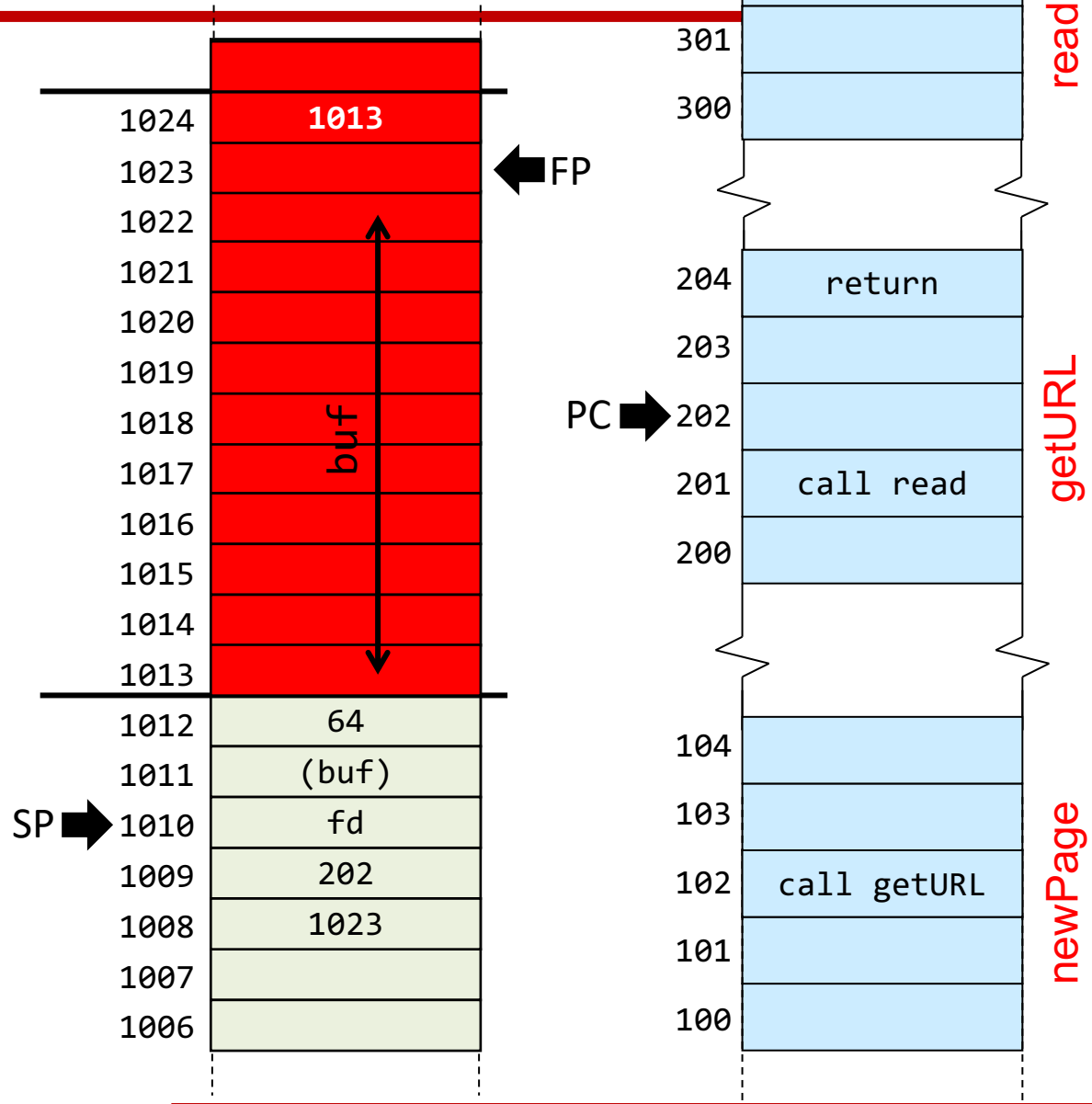
Mi a probléma?

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



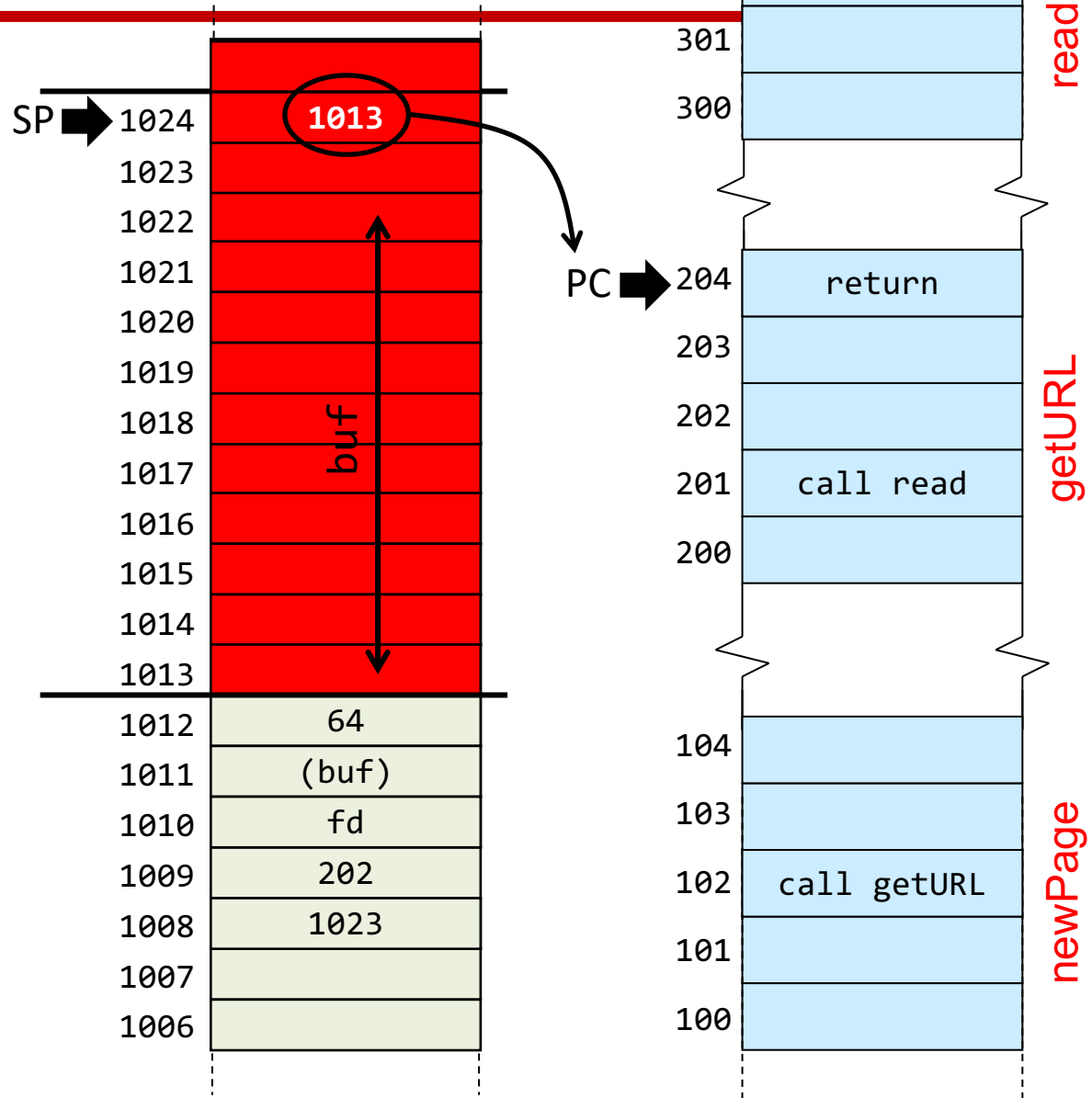
Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



Amikor a read() visszatér...

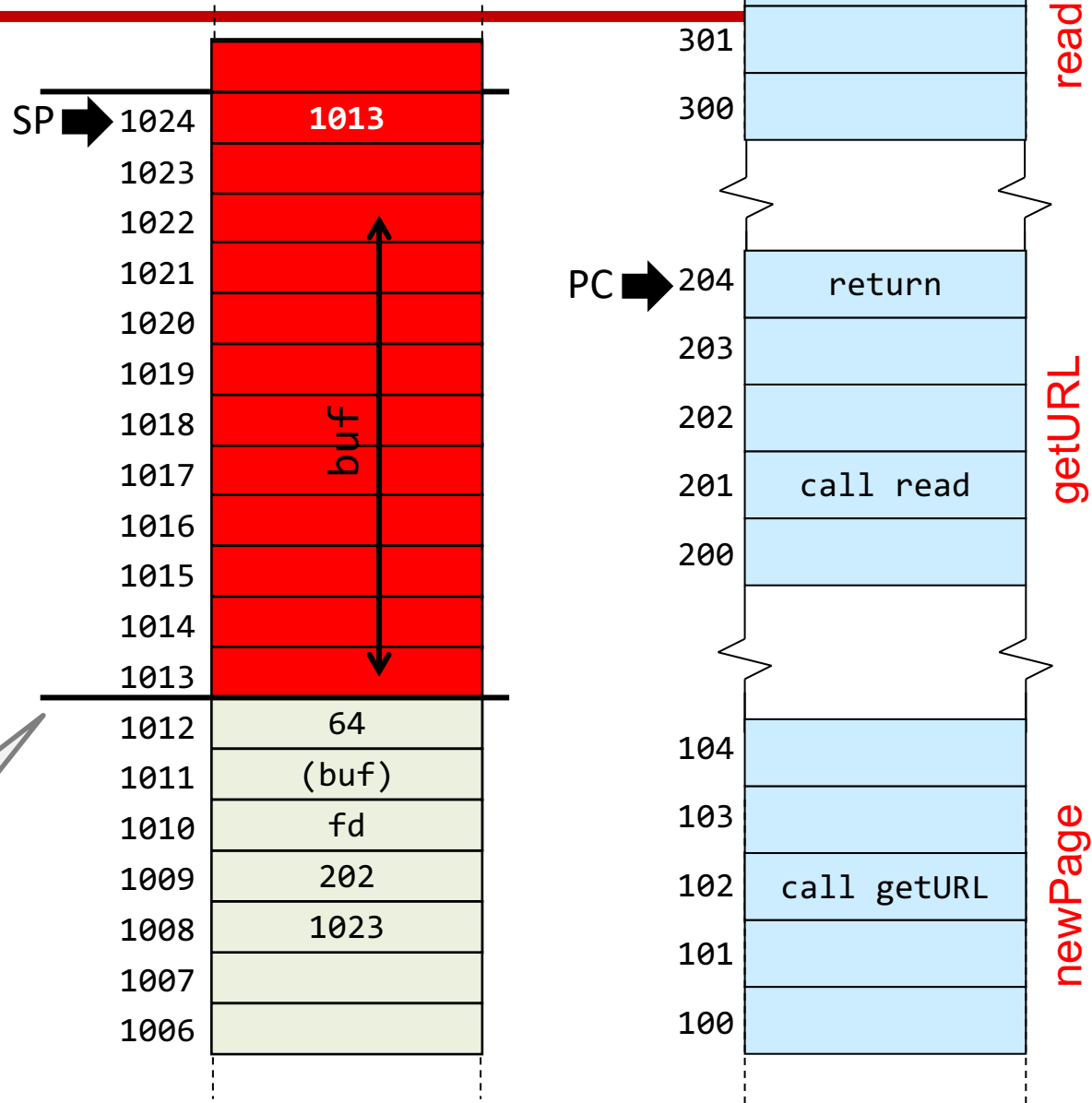
```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```



Amikor a read() visszatér...

```
getUrl()  
{  
    char buf[10];  
    read(keyboard, buf, 64);  
    loadPage(buf);  
}  
  
newPage()  
{  
    getUrl();  
}
```

... És ezután a processzor a támadó által definiált kódot fogja futtatni!





VÉDEKEZÉS

Hogyan lehet megállítani a támadást?

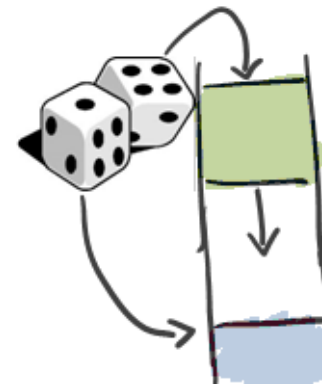
- A legjobb védekezés, ha megfelelő ellenőrzések vannak a kódban
 - Nagyon sok C/C++ fejlesztő van, és van közülük, aki elfelejti implementálni
 - Kövessük a biztonságos programozási elveket!
- Képes-e a rendszer védekezni valahogy?



DEP



canaries



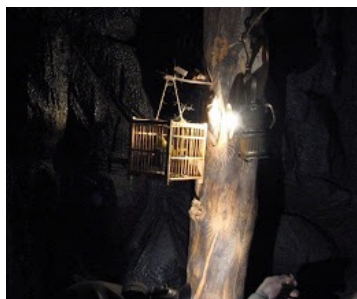
ASLR

DEP – Data Execution Prevention

- Cél: futtatható és írható memória területek elkülönítése
 - A stacknek írhatónak kell lennie, azonban nem szabad ott kódot futtatni
 - A programkódnak futtathatónak kell lennie azonban nem szabad írni
 - » Kivéve, ha modulokat dinamikusan kell betölteni
- Általában a memória kezelő egység (MMU) valósítja meg
 - Azok a lapok a memóriában, amelyek adatot tartalmaznak, beállíthatók, hogy ne legyenek futtathatók (NX bit)
- Az egész architektúra is támogathatja ezt a tervezéséből kifolyólag: Harvard architektúra
 - Fizikailag külön memóriában tárolja az adatot és a kódot
 - CPU képes olvasni a két memóriát egy időben
 - Az adat memória sosem fatható

Kanárik

- Régebben, a bányában használtak kanári madarakat, hogy észre vegyék, ha szénmonoxid van a levegőbe



- Stack kanári használható arra, hogy egy támadást észleljünk
- Egy 32 bites érték a visszatérési cím és a lokális változók között
 - Fordító segítségével alkalmazható ez a védelem
 - Újra kell fordítani a létező programokat ennek a használatához
- A függvény epilógusa ellenőrzi az értéket visszatérés előtt (hiba esetén leáll a program)



A kanári védelem is megkerülhető...

```
char gWelcome[] = "Welcome to our system! ";

void echo(int fd)
{
    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len);

    write(fd, "Type your name: ");
    read(fd, name, 128);

    memcpy(reply + len, name, 64);
    write(fd, reply, len + 64);
    return;
}

void server(int sockfd)
{
    while (1)
        echo(sockfd);
}
```

ret addr ow



reply

A kanári védelem is megkerülhető...

```
char gWelcome[] = "Welcome to our system! ";

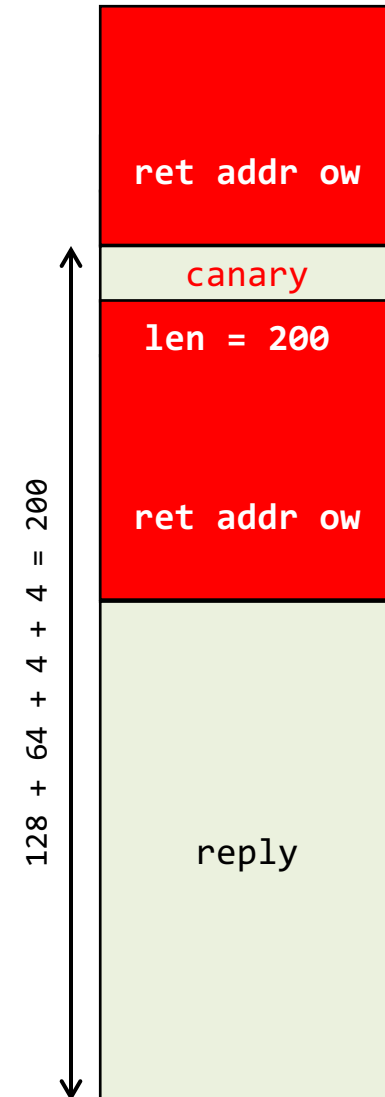
void echo(int fd)
{
    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len);

    write(fd, "Type your name: ");
    read(fd, name, 128);

    ➔ memcpy(reply + len, name, 64);
    write(fd, reply, len + 64);
    return;
}

void server(int sockfd)
{
    while (1)
        echo(sockfd);
}
```



- Address Space Layout Randomization
 - Az operációs rendszer véletlenszerű címekre tölti be az adat és program régiókat (stack, .data, .text, shared libraries)
 - Ezek a címek a program minden indításánál változnak
- Miért jó ez?
 - A támadó nem tud fix címeket használni a támadásában
 - » buffer overflow: a támadó nem tudja megtippelni a shellcode címét
 - » return-to-libc: a támadó nem tudja megtippelni a könyvtár függvény címét
 - » ROP: a támadó nem tudja megtippelni a gadgetek címét
- Elérhető implementáció az összes modern operációs rendszerben
 - Linux: kernel > 2.6.11
 - Windows: Vista
- A beágyazott rendszerek és IoT eszközök számára sajnos nem egyértelműen, hogy elérhető ez a védelem

Az ASLR története

- “the Linux PaX project first coined the term ASLR, and published the first design and implementation of ASLR in July 2001 as a patch for the Linux kernel”
- A PaX Teamet egy magyar „hacker” alapította: *pipacs*
 - **Itt végzett a BME-n!**
- pipacs/PaX Team nyerte meg 2011-ben a Pwnie lifetime achievement díjjat
 - A Pwnie Award célja, hogy kiemelkedő eredményeket díjazzon az IT biztonság területén
 - A győzteseket egy bizottság szavazati alapján választják
 - A díjat a Black Hat konferencián adják át minden évben



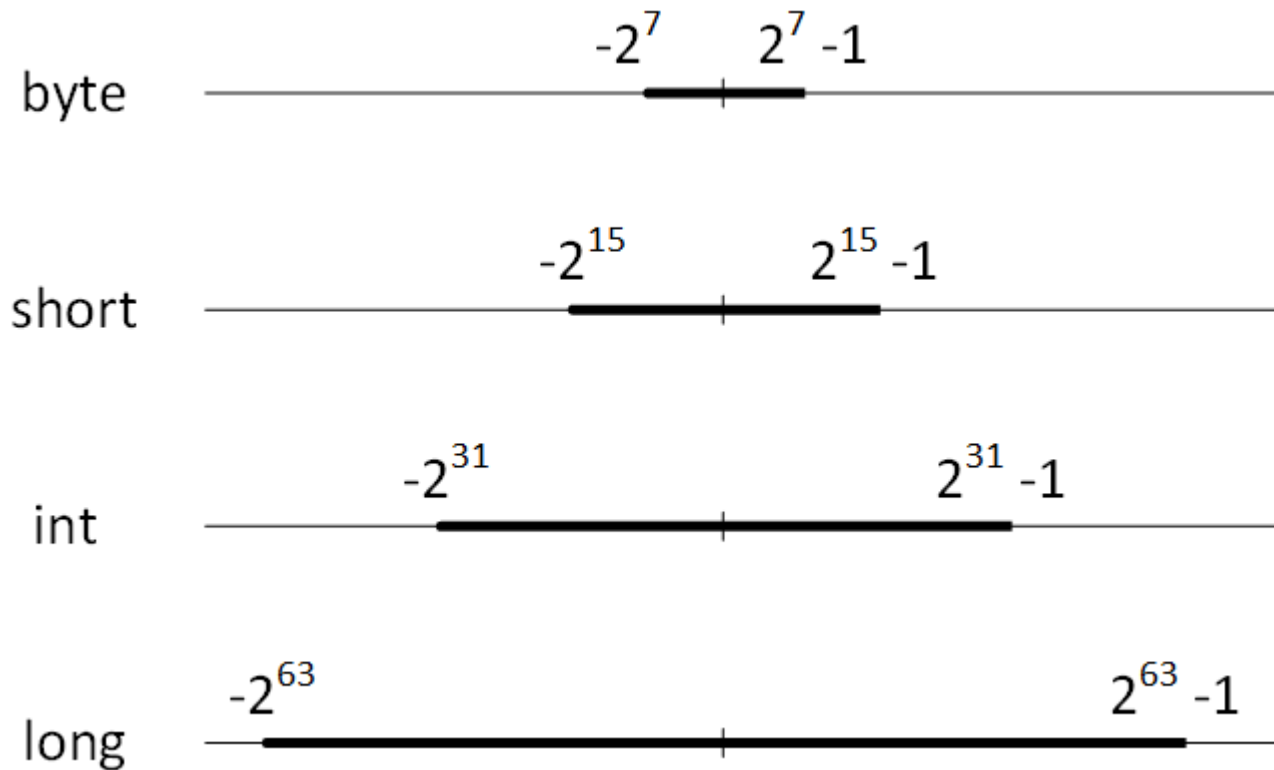
Az ASLR gyengesége

- Az ASLR egy 32 bites architektúrán limitált védelmet nyújt, a rendelkezésre álló randomizálható bitek száma miatt
 - Csak ~16 bit randomizálható a 32-ből
 - A védelem így egy brute force támadással percek alatt megkerülhető
- Az ASLR egy 64 bites architektúrán erős védelmet nyújt
 - 40 bit randomizálható a 64-ből
 - Ennyi esetnél is elképzelhető egy brute force támadás, azonban
 - » Jelentősen lassabban lehet eredményt elérni
 - » Nagy valószínűséggel detektálható
 - várhatóan sokszor összeomlik a támadás alatt a rendszer



INTEGER OVERFLOW

Számábrázolás



- Mi történik a tartományok szélén?

Probléma



```
1 int add(int a, int b) {  
2     return a+b;  
3 }
```

Probléma



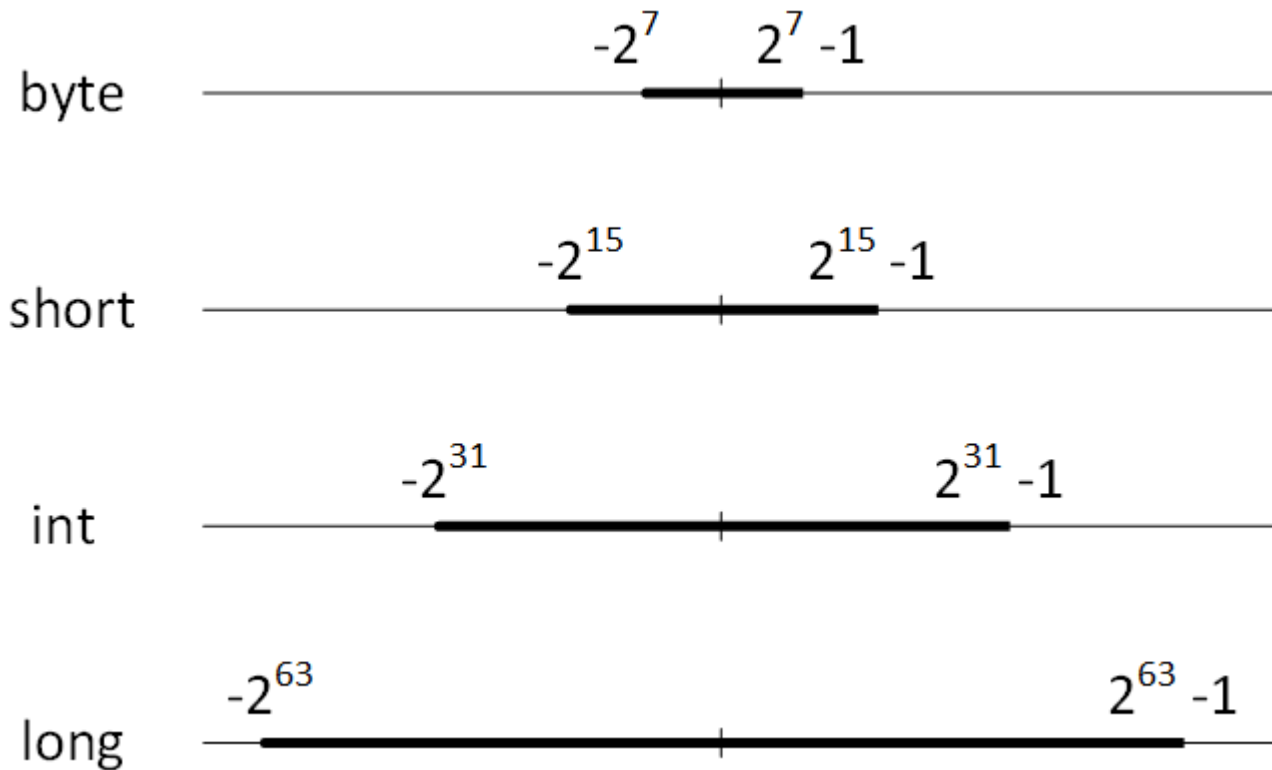
```
1 int add(int a, int b) {  
2     return a+b;  
3 }
```

a = INT_MAX

b = 1

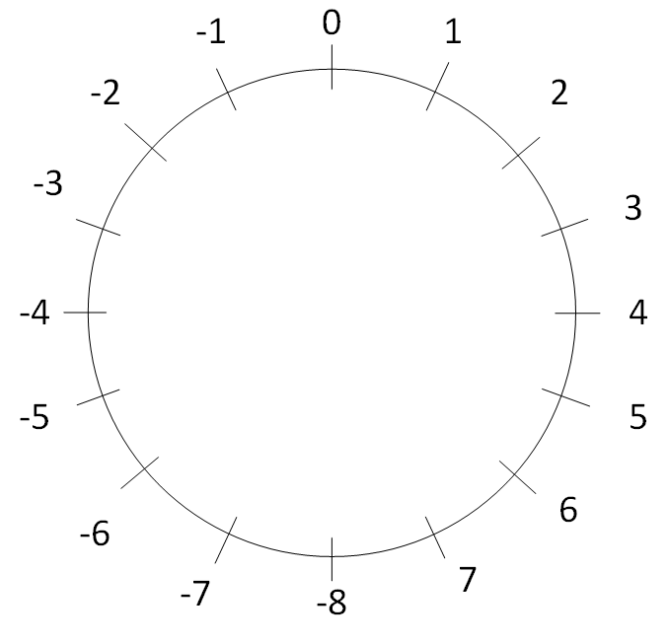
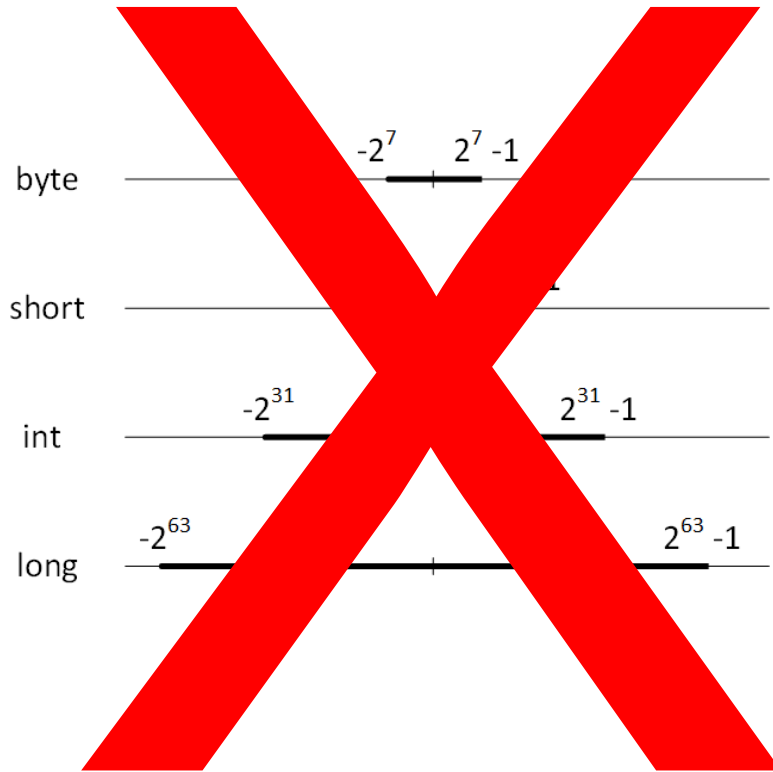
→ a+b = INT_MIN 😊

Számábrázolás



- Mi történik a tartományok szélén?
- Nincs garancia arra, hogy egy aritmetikai művelet matematikailag helyes eredményt ad.

Helyes megközelítés



Melyik nyelveket érinti?



Python



Java

C#

C / C++

...

Boeing 787 története

Latest in Boeing



Boeing is reportedly ending production of its 747 jumbo jet

🕒 07.04.20



Boeing successfully tests Starliner's parachutes ahead of second flight attempt

🕒 06.30.20



Boeing poised to start crucial 737 Max flight safety tests tomorrow

🕒 06.28.20

To keep a Boeing Dreamliner flying, reboot once every 248 days



Edgar Alvarez, @abcdedgar
May 1, 2015

27
Comments

13
Shares



AdChoices

The [787 Dreamliner](#) has been plagued with [battery woes](#) since its early days, to the point where the Federal Aviation Administration kept it [from](#)

Boeing 787 története



Fiora @ 日本語でFF14と.hack
@FioraAeterna



248 days == 2^{31} 100ths of a second.

even in 2015, our airplanes have integer overflow bugs



Ben CatchYourCough Goldacre  @bengoldacre

If you leave your Boeing 787 switched on for 248 days the power shuts off and you fall out of the sky. Epic bug. theguardian.com/business/2015/...

2:06 PM · May 1, 2015



489



1.1K people are Tweeting about this

Összefoglalás

- Sok témakört érintettünk:
 - Buffer overflow támadások
 - » Ötlete, gyakorlati kivitelezése
 - » Shellcode használata
 - Buffer overflow védekezés
 - » Kanárik, DEP, ASLR
 - » Nem védenek minden esetben!
 - Integer Overflow
- A statisztikák azt mutatják, hogy a buffer overflow támadások még jó ideig velünk maradnak
 - Beágyazott rendszereknél sok helyen használják a C nyelvet
 - Védekezési lehetőségek nem mindenhol elérhetőek
- Legjobb elkerülni a hibát a kódban!

További olvasnivalók

- Smashing The Stack For Fun And Profit - <https://insecure.org/stf/smashstack.html>
- Beyond stack smashing: recent advances in exploiting buffer overruns - <https://ieeexplore.ieee.org/document/1324594>
- Memory Errors: The Past, the Present, and the Future - <https://www.isg.rhul.ac.uk/sullivan/pubs/tr/technicalreport-ir-cs-73.pdf>
- 10Kstudents - General Introduction - <https://www.youtube.com/watch?v=0ytCmXJhMV4>
- SoK: Eternal War in Memory - <https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf>

Ellenőrző kérdések

- Melyik programozási nyelvek vannak leginkább kitéve a memória korrupciós hibáknak?
- Mi az a stack frame? Hol találhatóak a memóriában a függvény paraméterek és lokális változók?
- Hogyan működik egy stack overflow támadás?
- Mi lehet a célja egy támadónak a visszatérési cím módosításán kívül?
- A stack overflown kívül milyen más memória korrupciós hibákat ismer?
- Milyen védekezési módszereket ismer memória korrupciós támadásokkal szemben? Hogyan működnek ezek?