



Reasoning<sup>®</sup> Inspection Service

Reasoning  
Inspection Service  
Defect Data

Linux

v 2.4.19 Networking Sample

OPEN SOURCE

*19-Nov-2003*



P.O. Box 478

Menlo Park, CA 94026-0478

+1 650-324-2510

[www.reasoning.com](http://www.reasoning.com)

# Introduction

## Reasoning™ Inspection Services

As your development partner, Reasoning understands the challenges of adding functionality on time and within budget while maintaining quality and security standards.

To assist your development team, Reasoning's automated inspection services detect security vulnerabilities and reliability defects, providing many of the benefits of a manual code review. This is done as an outsourced service, with results available in less time and at lower cost than with traditional methods.

Reasoning inspects Java, C, and C++ code, providing 100% code coverage and finding vulnerabilities and defects that traditional QA processes often miss. Reasoning then delivers a detailed report that includes a complete description of each vulnerability or defect, the conditions under which it occurs, and the exact location within the code, eliminating time-consuming root-cause analysis. With Reasoning, developers become more effective at writing quality code.

## Types of Inspections

Reasoning provides two types of inspection services:

- The *Security Inspection Service* identifies vulnerabilities through which an attacker can gain unauthorized access or launch a denial of service attack.
- The *Reliability Inspection Service* identifies defects that can cause an application to crash or behave unpredictably.

## Deliverables

Reasoning delivers two types of reports at the conclusion of an inspection:

- *Data Reports* are designed for developers. It identifies the type and location of every vulnerability or defect found, and it describes the circumstances under which each will occur, allowing developers to rapidly evaluate and resolve the reported vulnerabilities or defects.
- *Metrics Reports* are designed for management. By highlighting problem areas within an application and providing industry comparisons and ratings, it enables managers to better plan development efforts.

This is the Reliability Defect Data report.

# SUMMARY DEFECT REPORT

---

## Inventory Summary

Reasoning inspected a complete application.

|  |                |
|--|----------------|
| Total Number of Source Files:                | 166            |
| Number of User Include Files:                | 651            |
| <b>Total Number of User Files Processed:</b> | <b>817</b>     |
| <br>   |                |
| Total LOC of Source Files:                   | 81,852         |
| Number LOC User Include Files:               | 43,650         |
| <b>Total LOC in Project:</b>                 | <b>125,502</b> |

## Defect Summary

The column *Defect Instances* in the table below details, per defect class, how many defects there are in the application.

The column *Files Affected* details, per defect class, the number of files in the application that have one or more defects.

| Inspection Class  | Defect Instances | Files Affected |
|---|------------------|----------------|
| Memory Leak<br><i>Reference to allocated memory is lost</i>                       | 1                | 1              |
| NULL Pointer Dereference<br><i>Expression dereferences a NULL pointer</i>         | 3                | 3              |
| Bad Deallocation<br><i>Deallocation is inappropriate for type of data</i>         | 0                | 0              |
| Out of Bounds Array Access<br><i>Expression accesses a value beyond the array</i> | 3                | 2              |
| Uninitialized Variable<br><i>Variable is not initialized prior to use</i>         | 1                | 1              |
| <b>Total Defect Instances</b>   | <b>8</b>         | <b>---</b>     |

## DETAILED DEFECT REPORT

---

|                      |  |
|----------------------|--|
| <b>DEFECT CLASS:</b> | Memory Leak  |
| <b>LOCATION:</b>     | src\linux-2.4.19\net\socket.c : 750  |
| <b>DESCRIPTION</b>   | Local variable <b>fna</b> , declared on line <b>735</b> , is assigned a pointer to a block of memory allocated by <b>kmalloc</b> on line <b>741</b> . No other pointer refers to this memory block, so it is inaccessible (still allocated, but unreachable) once <b>fna</b> goes out of scope after line <b>750</b> .   |
| <b>PRECONDITIONS</b> | The conditional expression ( <b>on</b> ) on line <b>739</b> evaluates to <b>true</b> AND<br>The conditional expression ( <b>fna==NULL</b> ) on line <b>742</b> evaluates to <b>false</b> AND<br>The conditional expression ( <b>(sk=sock-&gt;sk) == NULL</b> ) on line <b>749</b> evaluates to <b>true</b> .   |
| <b>IMPACT</b>        | Depending on how long the application runs, how frequently the leak occurs, and the amount of available (virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system. Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak. |

### CODE FRAGMENT

```

733 static int sock_fasync(int fd, struct file *filp, int on)
734 {
735     struct fasync_struct *fa, *fna=NULL, **prev;
736     struct socket *sock;
737     struct sock *sk;
738
739     if (on)
740     {
741         fna=(struct fasync_struct *)kmalloc(sizeof(struct
fasync_struct), GFP_KERNEL);
742         if(fna==NULL)
743             return -ENOMEM;
744     }
745
746     sock = socki_lookup(filp->f_dentry->d_inode);
747
748     if ((sk=sock->sk) == NULL)
749         return -EINVAL;
750
751     lock_sock(sk);
752
753     prev=&(sock->fasync_list);
754
755     for (fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
756         if (fa->fa_file==filp)
757             break;
758
759     if(on)
760
```

|                      |   |
|----------------------|---|
| <b>DEFECT CLASS:</b> | Null Pointer Dereference  |
| <b>LOCATION:</b>     | src\linux-2.4.19\net\ipv4\ip_options.c : 262  |
| <b>DESCRIPTION</b>   | The local pointer variable <b>skb</b> , passed in as an argument on line <b>245</b> , may be NULL where it is dereferenced on line <b>262</b> .   |
| <b>PRECONDITIONS</b> | The conditional expression <b>skb</b> on line <b>252</b> evaluates to <b>false</b> AND<br>The conditional expression <b>(!opt)</b> on line <b>254</b> evaluates to <b>false</b> AND<br>The conditional expression <b>opt-&gt;is_data</b> on line <b>262</b> evaluates to <b>false</b> . |
| <b>IMPACT</b>        | A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.  |

**CODE FRAGMENT**

```

245 int ip_options_compile(struct ip_options * opt, struct sk_buff * skb)
246 {
...
252     struct rtable *rt = skb ? (struct rtable*)skb->dst : NULL;
253
254     if (!opt) {
255         opt = &(IPCB(skb)->opt);
256         memset(opt, 0, sizeof(struct ip_options));
257         iph = skb->nh.raw;
258         opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct
iphdr);
259         optptr = iph + sizeof(struct iphdr);
260         opt->is_data = 0;
261     } else {
262         optptr = opt->is_data ? opt->__data : (unsigned char*)&(skb-
>nh.iph[1]);
263         iph = optptr - sizeof(struct iphdr);
264     }
265
266     for (l = opt->optlen; l > 0; ) {
267         switch (*optptr) {

```

|                      |  |
|----------------------|--|
| <b>DEFECT CLASS:</b> | Null Pointer Dereference   |
| <b>LOCATION:</b>     | src\linux-2.4.19\net\sched\cls_rsvp.h : 526  |
| <b>DESCRIPTION</b>   | The local pointer variable <b>pinfo</b> , declared on line <b>429</b> , and assigned on line <b>429</b> , may be NULL where it is dereferenced on line <b>526</b> . Similar errors can be found on lines 533, 568, 569 and 570.  |
| <b>PRECONDITIONS</b> | The conditional expression <b>(tb[TCA_RSVP_PINFO-1])</b> on line <b>488</b> evaluates to <b>false</b> AND<br>The conditional expression <b>((f-&gt;handle = gen_handle(tp, h1   (h2&lt;&lt;8))) == 0)</b> on line <b>510</b> evaluates to <b>false</b> AND<br>The conditional expression <b>(f-&gt;tunnelhdr)</b> on line <b>513</b> evaluates to <b>false</b> AND<br>The for loop on line <b>524</b> is executed with <b>(s=*sp) != NULL</b> evaluates to <b>true</b> AND<br>The conditional expression <b>dst[RSVP_DST_LEN-1] == s-&gt;dst[RSVP_DST_LEN-1]</b> on line <b>525</b> evaluates to <b>true</b> . |
| <b>IMPACT</b>        | A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.   |

**CODE FRAGMENT**

```

421 static int rsvp_change(struct tcf_proto *tp, unsigned long base,
422                       u32 handle,
423                       struct rtattr **tca,
424                       unsigned long *arg)
425 {
426     .
427     .
428     .
429     struct tc_rsvp_pinfo *pinfo = NULL;
430     .
431     .
432     .
433     .
434     .
435     .
436     .
437     .
438     .
439     .
440     .
441     .
442     .
443     .
444     .
445     .
446     .
447     .
448     .
449     .
450     .
451     .
452     .
453     .
454     .
455     .
456     .
457     .
458     .
459     .
460     .
461     .
462     .
463     .
464     .
465     .
466     .
467     .
468     .
469     .
470     .
471     .
472     .
473     .
474     .
475     .
476     .
477     .
478     .
479     .
480     .
481     .
482     .
483     .
484     .
485     .
486     .
487     .
488     if (tb[TCA_RSVP_PINFO-1]) {
489         .
490         .
491         .
492         .
493         .
494         .
495         .
496         .
497         .
498         .
499         .
500         .
501         .
502         .
503         .
504         .
505         .
506         .
507         .
508         .
509         .
510         .
511         .
512         .
513         .
514         .
515         .
516         .
517         .
518         .
519         .
520         .
521         .
522         .
523         .
524         .
525         .
526         .
527         .
528         .
529         .
530         .
531         .
532         .
533         .
534         .
535         .
536         .
537         .
538         .
539         .
540         .
541         .
542         .
543         .
544         .
545         .
546         .
547         .
548         .
549         .
550         .
551         .
552         .
553         .
554         .
555         .
556         .
557         .
558         .
559         .
560         .
561         .
562         .
563         .
564         .
565         .
566         .
567         .
568         .
569         .
570         .
571         .
572         .
573         .
574         .
575         .
576         .
577         .
578         .
579         .
580         .
581         .
582         .
583         .
584         .
585         .
586         .
587         .
588         .
589         .
590         .
591         .
592         .
593         .
594         .
595         .
596         .
597         .
598         .
599         .
600         .
601         .
602         .
603         .
604         .
605         .
606         .
607         .
608         .
609         .
610         .
611         .
612         .
613         .
614         .
615         .
616         .
617         .
618         .
619         .
620         .
621         .
622         .
623         .
624         .
625         .
626         .
627         .
628         .
629         .
630         .
631         .
632         .
633         .
634         .
635         .
636         .
637         .
638         .
639         .
640         .
641         .
642         .
643         .
644         .
645         .
646         .
647         .
648         .
649         .
650         .
651         .
652         .
653         .
654         .
655         .
656         .
657         .
658         .
659         .
660         .
661         .
662         .
663         .
664         .
665         .
666         .
667         .
668         .
669         .
670         .
671         .
672         .
673         .
674         .
675         .
676         .
677         .
678         .
679         .
680         .
681         .
682         .
683         .
684         .
685         .
686         .
687         .
688         .
689         .
690         .
691         .
692         .
693         .
694         .
695         .
696         .
697         .
698         .
699         .
700         .
701         .
702         .
703         .
704         .
705         .
706         .
707         .
708         .
709         .
710         .
711         .
712         .
713         .
714         .
715         .
716         .
717         .
718         .
719         .
720         .
721         .
722         .
723         .
724         .
725         .
726         .
727         .
728         .
729         .
730         .
731         .
732         .
733         .
734         .
735         .
736         .
737         .
738         .
739         .
740         .
741         .
742         .
743         .
744         .
745         .
746         .
747         .
748         .
749         .
750         .
751         .
752         .
753         .
754         .
755         .
756         .
757         .
758         .
759         .
760         .
761         .
762         .
763         .
764         .
765         .
766         .
767         .
768         .
769         .
770         .
771         .
772         .
773         .
774         .
775         .
776         .
777         .
778         .
779         .
780         .
781         .
782         .
783         .
784         .
785         .
786         .
787         .
788         .
789         .
790         .
791         .
792         .
793         .
794         .
795         .
796         .
797         .
798         .
799         .
800         .
801         .
802         .
803         .
804         .
805         .
806         .
807         .
808         .
809         .
810         .
811         .
812         .
813         .
814         .
815         .
816         .
817         .
818         .
819         .
820         .
821         .
822         .
823         .
824         .
825         .
826         .
827         .
828         .
829         .
830         .
831         .
832         .
833         .
834         .
835         .
836         .
837         .
838         .
839         .
840         .
841         .
842         .
843         .
844         .
845         .
846         .
847         .
848         .
849         .
850         .
851         .
852         .
853         .
854         .
855         .
856         .
857         .
858         .
859         .
860         .
861         .
862         .
863         .
864         .
865         .
866         .
867         .
868         .
869         .
870         .
871         .
872         .
873         .
874         .
875         .
876         .
877         .
878         .
879         .
880         .
881         .
882         .
883         .
884         .
885         .
886         .
887         .
888         .
889         .
890         .
891         .
892         .
893         .
894         .
895         .
896         .
897         .
898         .
899         .
900         .
901         .
902         .
903         .
904         .
905         .
906         .
907         .
908         .
909         .
910         .
911         .
912         .
913         .
914         .
915         .
916         .
917         .
918         .
919         .
920         .
921         .
922         .
923         .
924         .
925         .
926         .
927         .
928         .
929         .
930         .
931         .
932         .
933         .
934         .
935         .
936         .
937         .
938         .
939         .
940         .
941         .
942         .
943         .
944         .
945         .
946         .
947         .
948         .
949         .
950         .
951         .
952         .
953         .
954         .
955         .
956         .
957         .
958         .
959         .
960         .
961         .
962         .
963         .
964         .
965         .
966         .
967         .
968         .
969         .
970         .
971         .
972         .
973         .
974         .
975         .
976         .
977         .
978         .
979         .
980         .
981         .
982         .
983         .
984         .
985         .
986         .
987         .
988         .
989         .
990         .
991         .
992         .
993         .
994         .
995         .
996         .
997         .
998         .
999         .
1000        .

```

```
534  
535 insert:                /* OK, we found appropriate session */  
536
```

|                      |   |
|----------------------|---|
| <b>DEFECT CLASS:</b> | Null Pointer Dereference  |
| <b>LOCATION:</b>     | src/linux-2.4.19/net/sunrpc/xprt.c : 435  |
| <b>DESCRIPTION</b>   | The local pointer variable <b>sock</b> , declared on line <b>400</b> , and assigned on line <b>400</b> , may be NULL where it is dereferenced on line <b>435</b> .  |
| <b>PRECONDITIONS</b> | The conditional expression ( <b>!xprt-&gt;addr.sin_port</b> ) on line <b>412</b> evaluates to <b>false</b> AND<br>The conditional expression ( <b>!xprt_lock_write(xprt, task)</b> ) on line <b>417</b> evaluates to <b>false</b> AND<br>The conditional expression ( <b>xprt_connected(xprt)</b> ) on line <b>419</b> evaluates to <b>false</b> AND<br>The conditional expression <b>sock</b> on line <b>422</b> evaluates to <b>false</b> AND<br>The conditional expression ( <b>!(inet = xprt-&gt;inet)</b> ) on line <b>425</b> evaluates to <b>false</b> . |
| <b>IMPACT</b>        | A NULL pointer dereference usually causes a program exception. Notable exceptions to this rule are some embedded environments, in which a NULL pointer dereference does not cause program termination.  |

**CODE FRAGMENT**

```

396 void
397 xprt_reconnect(struct rpc_task *task)
398 {
...
400     struct socket    *sock = xprt->sock;
...
412     if (!xprt->addr.sin_port) {
413         task->tk_status = -EIO;
414         return;
415     }
416
417     if (!xprt_lock_write(xprt, task))
418         return;
419     if (xprt_connected(xprt))
420         goto out_write;
421
422     if (sock && sock->state != SS_UNCONNECTED)
423         xprt_close(xprt);
424     status = -ENOTCONN;
425     if (!(inet = xprt->inet)) {
426         /* Create an unconnected socket */
427         if (!(sock = xprt_create_socket(xprt->prot, &xprt->timeout)))
428             goto defer;
429         xprt_bind_socket(xprt, sock);
430         inet = sock->sk;
431     }
432
433     /* Now connect it asynchronously. */
434     dprintk("RPC: %4d connecting new socket\n", task->tk_pid);
435     status = sock->ops->connect(sock, (struct sockaddr *) &xprt->addr,
436                               sizeof(xprt->addr), O_NONBLOCK);
437
438     if (status < 0) {
439         switch (status) {
440             case -EALREADY:
441             case -EINPROGRESS:
442                 status = 0;
443                 break;
444             case -EISCONN:

```

**DEFECT CLASS:** Out Of Bounds Array Access

**LOCATION:** src/linux-2.4.19/net/core/rtnetlink.c : 311

**DESCRIPTION** The array index on line **311** may be **NPROTO**, outside the bounds of the array **rtnetlink\_links**. The valid indices for the array **rtnetlink\_links**, declared on line **81**, range from 0 to **NPROTO-1**.

**PRECONDITIONS** The conditional expression **(!(nlh->nlmsg\_flags&NLM\_F\_REQUEST))** on line **286** evaluates to **false**.

The conditional expression **(type < RTM\_BASE)** on line **292** evaluates to **false**.

The conditional expression **(type > RTM\_MAX)** on line **296** evaluates to **false**.

The conditional expression **(nlh->nlmsg\_len < NLMSG\_LENGTH(sizeof(struct rtgenmsg)))** on line **302** evaluates to **false**.

The conditional expression **(family > NPROTO)** on line **306** evaluates to **false**.

**IMPACT** An out-of-bounds array access defect can cause data corruption or lead to a program exception.

**CODE FRAGMENT**

```

81  struct rtnetlink_link * rtnetlink_links[NPROTO];
...
271  static __inline__ int
272  rtnetlink_rcv_msg(struct sk_buff *skb, struct nlmsg_hdr *nlh, int *errp)
273  {
274      struct rtnetlink_link *link;
275      struct rtnetlink_link *link_tab;
276      struct rtattr *rta[RTATTR_MAX];
277
278      int exclusive = 0;
279      int sz_idx, kind;
280      int min_len;
281      int family;
282      int type;
283      int err;
284
285      /* Only requests are handled by kernel now */
286      if (!(nlh->nlmsg_flags&NLM_F_REQUEST))
287          return 0;
288
289      type = nlh->nlmsg_type;
290
291      /* A control message: ignore them */
292      if (type < RTM_BASE)
293          return 0;
294
295      /* Unknown message: reply with EINVAL */
296      if (type > RTM_MAX)
297          goto err_inval;
298
299      type -= RTM_BASE;
300
301      /* All the messages must have at least 1 byte length */
302      if (nlh->nlmsg_len < NLMSG_LENGTH(sizeof(struct rtgenmsg)))
303          return 0;
304
305      family = ((struct rtgenmsg*)NLMSG_DATA(nlh))->rtgen_family;
306      if (family > NPROTO) {
307          *errp = -EAFNOSUPPORT;
308          return -1;
309      }
310
311      link_tab = rtnetlink_links[family];
312      if (link_tab == NULL)
313          link_tab = rtnetlink_links[PF_UNSPEC];

```

```
314     link = &link_tab[type];
315
316     sz_idx = type>>2;
317     kind = type&3;
318
319     if (kind != 2 && !cap_raised(NETLINK_CB(skb).eff_cap, CAP_NET_ADMIN))
320     {
321         *errp = -EPERM;
322         return -1;
323     }
324 }
```

**DEFECT CLASS:** Out Of Bounds Array Access

**LOCATION:** src/linux-2.4.19/net/sched/sch\_gred.c : 339

**DESCRIPTION** The array index on line 339, `TCA_GRED_DPS-1`, which evaluates to 2, may be outside the bounds of the array `tb`. The valid indices for the array `tb`, declared on line 330, range from 0 to `TCA_GRED_STAB-1`, which evaluates to 1. A similar error can be found on line 481.

**PRECONDITIONS** The conditional expression `(opt == NULL || rtattr_parse(tb, TCA_GRED_STAB, RTA_DATA(opt), RTA_PAYLOAD(opt)))` on line 334 evaluates to `false` AND  
The conditional expression `tb[TCA_GRED_PARAMS-1] == 0 && tb[TCA_GRED_STAB-1] == 0` on line 338 evaluates to `true`.

**IMPACT** An out-of-bounds array access defect can cause data corruption or lead to a program exception.

#### CODE FRAGMENT

```

324 static int gred_change(struct Qdisc *sch, struct rtattr *opt)
325 {
326     struct gred_sched *table = (struct gred_sched *)sch->data;
327     struct gred_sched_data *q;
328     struct tc_gred_qopt *ctl;
329     struct tc_gred_sopt *sopt;
330     struct rtattr *tb[TCA_GRED_STAB];
331     struct rtattr *tb2[TCA_GRED_STAB];
332     int i;
333
334     if (opt == NULL ||
335         rtattr_parse(tb, TCA_GRED_STAB, RTA_DATA(opt),
336                     RTA_PAYLOAD(opt)) )
337         return -EINVAL;
338
339     if (tb[TCA_GRED_PARAMS-1] == 0 && tb[TCA_GRED_STAB-1] == 0 &&
340         tb[TCA_GRED_DPS-1] != 0) {
341         rtattr_parse(tb2, TCA_GRED_DPS, RTA_DATA(opt),
342                     RTA_PAYLOAD(opt));
343
344         sopt = RTA_DATA(tb2[TCA_GRED_DPS-1]);
345         table->Dps=sopt->Dps;
346         table->def=sopt->def_DP;
347         table->grio=sopt->grio;
348         table->initd=0;
349         /* probably need to clear all the table DP entries as well */
350         MOD_INC_USE_COUNT;

```

**DEFECT CLASS:** Out Of Bounds Array Access

**LOCATION:** src\linux-2.4.19\net\sched\sched\_gred.c : 343

**DESCRIPTION** The array index on line 343, `TCA_GRED_DPS-1`, which evaluates to 2, may be outside the bounds of the array `tb2`. The valid indices for the array `tb2`, declared on line 331, range from 0 to `TCA_GRED_STAB-1`, which evaluates to 1. A similar error can be found on line 484.

**PRECONDITIONS** The conditional expression `(opt == NULL || rtattr_parse(tb, TCA_GRED_STAB, RTA_DATA(opt), RTA_PAYLOAD(opt)))` on line 334 evaluates to `false` AND The conditional expression `(tb[TCA_GRED_PARAMS-1] == 0 && tb[TCA_GRED_STAB-1] == 0 && tb[TCA_GRED_DPS-1] != 0)` on line 338 evaluates to `true`.

**IMPACT** An out-of-bounds array access defect can cause data corruption or lead to a program exception.

**CODE FRAGMENT**

```

324 static int gred_change(struct Qdisc *sch, struct rtattr *opt)
325 {
326     struct gred_sched *table = (struct gred_sched *)sch->data;
327     struct gred_sched_data *q;
328     struct tc_gred_qopt *ctl;
329     struct tc_gred_sopt *sopt;
330     struct rtattr *tb[TCA_GRED_STAB];
331     struct rtattr *tb2[TCA_GRED_STAB];
332     int i;
333
334     if (opt == NULL ||
335         rtattr_parse(tb, TCA_GRED_STAB, RTA_DATA(opt),
336                     RTA_PAYLOAD(opt)) )
337         return -EINVAL;
338
339     if (tb[TCA_GRED_PARAMS-1] == 0 && tb[TCA_GRED_STAB-1] == 0 &&
340         tb[TCA_GRED_DPS-1] != 0) {
341         rtattr_parse(tb2, TCA_GRED_DPS, RTA_DATA(opt),
342                     RTA_PAYLOAD(opt));
343
344         sopt = RTA_DATA(tb2[TCA_GRED_DPS-1]);
345         table->Dps=sopt->Dps;
346         table->def=sopt->def_DP;
347         table->grio=sopt->grio;
348         table->initd=0;
349         /* probably need to clear all the table DP entries as well */
350         MOD_INC_USE_COUNT;
351         return 0;
352     }

```

|                      |  |
|----------------------|--|
| <b>DEFECT CLASS:</b> | Uninitialized Variable   |
| <b>LOCATION:</b>     | src/linux-2.4.19/net/core/filter.c : 312   |
| <b>DESCRIPTION</b>   | The local variable <b>mem</b> , declared on line <b>79</b> , is used on line <b>312</b> , before <b>mem</b> has been initialized. A similar error can be found on line 316.            |
| <b>PRECONDITIONS</b> | The for loop on line <b>87</b> is executed with <b>pc &lt; flen</b> evaluates to <b>true</b> AND<br>The case statement <b>BPF_LD BPF_MEM</b> on line <b>311</b> is executed.           |
| <b>IMPACT</b>        | Usage of uninitialized variables can cause unpredictable results in the program (because the value of the variable is essentially random), and in the worst case, a program exception. |

**CODE FRAGMENT**

```

69 int sk_run_filter(struct sk_buff *skb, struct sock_filter *filter, int flen)
70 {
71     unsigned char *data = skb->data;
72     /* len is UNSIGNED. Byte wide insns relies only on implicit
73        type casts to prevent reading arbitrary memory locations.
74     */
75     unsigned int len = skb->len-skb->data_len;
76     struct sock_filter *fentry;          /* We walk down these */
77     u32 A = 0;                          /* Accumulator */
78     u32 X = 0;                          /* Index Register */
79     u32 mem[BPF_MEMWORDS];             /* Scratch Memory Store */
80     int k;
81     int pc;
82
83     /*
84     * Process array of filter instructions.
85     */
86
87     for(pc = 0; pc < flen; pc++)
88     {
89         fentry = &filter[pc];
90
91         switch(fentry->code)
92         {
93             . . .
94
95             case BPF_LD|BPF_MEM:
96                 A = mem[fentry->k];
97                 continue;
98
99             case BPF_LDX|BPF_MEM:
100                X = mem[fentry->k];
101                continue;
102
103             case BPF_MISC|BPF_TAX:
104                 X = A;
105                 continue;
106
107             . . .

```

## A. UNDERSTANDING DEFECT CLASS DESCRIPTIONS

---

This Appendix provides a detailed explanation of each of the Illuma defect classes for C/C++. These examples assist the client in evaluating the impact of each of the defects listed in the Detailed Defect Report.

For each defect class, this document provides:

1. a short description of the class;
2. the likely impact of the defect;
3. advice on how the defect can be repaired;
4. an example code fragment that explains the defect in detail. Note that the example code fragments are *not* from the inspected application.

In general, *data corruption* is considered the worst impact. Data corruption can go unnoticed for weeks while damage continues to accumulate.

When a *program exception* occurs, there is better evidence that something went wrong. Even in this case, the failure sometimes goes unnoticed, for example if the program is a cgi-bin program that runs frequently for a short period of time. However, in most other situations, a program exception is a fatal error that leads to immediate program termination. In an embedded system or "daemon" process, such a failure could be catastrophic for the overall system.

When *unpredictable results* happen, they may eventually result in data corruption and/or program exceptions, but they may also go unnoticed. If they finally cause an observable error to occur, a large effort is usually required to track down exactly where the error occurs, because the reduction process to track the error down often makes the error fail to recur. Certain defects in one application may cause other programs on the same platform, or even the operating system, to fail, often after some time has passed. These defects are particularly difficult to track down.

*Risk* is a qualitative measure of the hazard associated with a particular defect. High-risk defects are likely to cause data corruption or program exceptions. Low-risk defects may cause performance degradation but are otherwise unlikely to have a serious impact. Application-dependent risks must be evaluated by taking into account additional information about the compiler, runtime environment, or actual usage pattern of the application.

Finally, over 70% of the effort spent on most C/C++ applications is spent in maintenance. Even when not directly preventing failures, the removal of defects may greatly reduce the cost of:

- extending the system with new functionality, such as web-enabling it;
- finding the root cause of a failure;
- training new staff and consultants; and
- rewriting or replacing the system.

In the descriptions of the defect classes, as much context as reasonably possible is provided. It is beyond the scope of this document, however, to explain everything that is needed to understand fully how to program in C/C++. The following publications may help to gain a deeper understanding:

- Kernighan & Ritchie, *The C Programming Language*, 2nd Edition, Bell Telephone Laboratories, Inc., 1988.
- Bjarne Stroustrup, *The C++ Programming Language*, 3rd Edition, AT&T, 1997.
- Steve McConnell, *Code Complete*, Microsoft Press, 1993.
- Andrew Koenig, *C Traps and Pitfalls*, AT&T Bell Telephone Labs, 1989.
- P.J. Plauger, *The Standard C library*, Prentice Hall, 1992.
- Scott Meyers, *Effective C++*, 2nd Edition, Addison Wesley, 1998.

## Memory Leak

### DESCRIPTION

*Memory leak* refers to the loss of available memory space that occurs when dynamic data (memory allocated on the heap by calling any of the standard C library routines `malloc()`, `calloc()`, `realloc()`, `strdup()` or the C++ operator `new`) is no longer used but never deallocated (by calling `free()`, `realloc()` or the C++ operator `delete`).

### IMPACT

Each time the leak occurs, the application drains the available memory pool. On some systems, memory may be allocated from a global pool, in which case the loss of available memory may affect the entire system, not just the application that caused it. Even on virtual-memory systems, where each process has its own protected address space, the gradual increase in application size can result in performance degradation that affects the entire system.

Depending on how long the application runs, how frequently the leak occurs, and the amount of available (including virtual) memory, memory leaks will sooner or later cause performance degradation of the application, and potentially of the entire system.

Eventually, the performance degradation may lead to a fatal out-of-memory condition. This condition may be encountered by an application unrelated to the one that caused the memory leak.

Understanding the application is important to rating this defect. For example, if this defect occurs in a daemon, the impact is almost always high, but if it occurs in a CGI script, the impact is usually negligible.

### REPAIR

A deallocation corresponding to each allocation is not always necessary, but it is a safe programming practice. For example, it is not necessary to deallocate dynamic data that is allocated in `main()`, since all dynamic data is freed after `main()` finishes, but a conservative programmer might include the deallocation anyway.

### EXAMPLE

```
int check_msg() {
    msg_t *msg;
    int status = RET_FAIL;
    ...
    msg = (msg_t *)calloc(1, sizeof(*msg));
    if (!msg) {
        errno = OUTFOFMEMORY;
        return status;
    }
    err = Readmsg(msg);
    if (err == -1)
        return status;
    ...
}
```

In this example, memory space is allocated for a new `msg`. One possible execution path, however, is where the `Readmsg()` call fails; this leads to a return without the memory

allocated for `msg` being freed. If this function is called frequently, and `Readmsg()` frequently fails, this could result in a large number of memory leaks. Eventually, this can lead to a fatal out-of-memory program exception.

## NULL Pointer Dereference

### DESCRIPTION

A `NULL` pointer is a pointer that refers to a specific, invalid memory address. A *dereference* means following a pointer to the memory location it refers to and accessing the data at that location. Thus, a *NULL pointer dereference* refers to an attempt to access data at this invalid address.

This defect class also reports situations where a `NULL` pointer is used in an assignment. Even though the assignment is not a defect by itself, a subsequent dereference will cause a program exception. The defect is reported at the assignment, because this is the earliest point in the code where the problem could be identified.

### IMPACT

On most general-purpose computing platforms (such as Windows or UNIX), a `NULL` pointer dereference usually causes a program exception. On systems without memory management hardware, such references may not be detected at all.

### REPAIR

To repair these defects, an if-statement checking for `NULL` values should be placed around the statements that dereference the pointers. Appropriate error-recovery should also be provided for the situations where the pointers are `NULL`.

### EXAMPLE

```
char* ptr = strrchr(tmp_eq, '-');  
int chan_num = atoi(ptr+1);
```

The string function `strrchr()` returns a pointer to the last occurrence of '-' in the string `tmp_eq`, or `NULL` if the character is not present. The next line of code will cause a program exception if `ptr` is `NULL`, because `atoi()` does not accept a `NULL` pointer. A check should be performed before the call to `atoi()` to ensure that `ptr` is not `NULL`.

## Bad Deallocation

### DESCRIPTION

*Bad deallocation* refers to the use of an inappropriate memory release operation (standard C library routines `free()` or `realloc()`, or C++ operators `delete` or `delete[]`) for deallocating memory, or to the deallocation of memory that was never explicitly allocated.

### IMPACT

Depending on the compiler and the specific operation, the impact of this defect may range from no effect at all, to unexpected results, a memory leak, memory corruption, or a program exception.

Specifically:

- The use of `delete` on memory allocated with `new[]` may be a memory leak, because only the first element of the array is released, not the entire array;
- The use of `delete` on memory allocated with `malloc()`, `calloc()` or `realloc()`, or `strdup()` may cause memory corruption, or a program exception;
- The use of `free()` or `realloc()` on memory allocated with `new` may cause memory corruption, or a program exception;
- The use of `free()` or `realloc()` on memory that is not heap-allocated may cause memory corruption, or a program exception.

### REPAIR

Memory allocated with `malloc()`, `calloc()`, `realloc()`, or `strdup()`, should be deallocated only with `free()` or `realloc()`. Similarly, memory allocated with `new` should be deallocated with `delete`, and memory allocated with `new []` should be deallocated with `delete []`. Furthermore, `free()` and `delete` should be applied to the exact same pointer value that was returned by the corresponding allocating expression.

### EXAMPLE 1

```
DS3_NOC noct3msg;
memset((void *)&noct3msg, 0, sizeof(DS3_NOC));
...
delete (&noct3msg);
```

In this example, a local variable is initialized to all 0's using the `memset()` function. One of the rules in C++ is that the `delete` operator can only be applied to a pointer value that was previously obtained from the `new` operator. Use of the `delete` operator in the Example 1 will result in a program exception.

**EXAMPLE 2**

```
char* buffer = new char[100];
...
buffer++;
delete [] buffer;
```

In this example, a new character array called `buffer` is allocated, and later the base pointer to that array is incremented. When the memory is deallocated, `buffer` no longer points to the beginning of the allocated block. This will often lead to a program exception, or corruption of the heap.

**EXAMPLE 3**

```
char* p = new char[100];
...
delete p;
```

The problem in this example is the missing `[]` on the `delete` operator. Depending on the compiler, this can result in a program exception, or corruption of the heap.

**EXAMPLE 4**

```
void parse_message(char *msg) {
    char formatstring[100];
    ...
    free(formatstring);
    return;
}
```

The character array `formatstring` in this example is stack-allocated. It is inappropriate to call `free()` on stack-allocated memory. Depending on the memory manager, this can result in a program exception, or corruption of the heap.

## Out of Bounds Array Access

### DESCRIPTION

An *out-of-bounds array access* refers to a defect where an array index expression is not within the upper and lower bounds of the array.

### IMPACT

An out-of-bounds array access defect can cause data corruption or lead to a program exception.

### REPAIR

Array indexing needs to be guarded against out-of-bounds defects.

For situations where one array is copied into another, this type of defect can be repaired by adjusting array sizes.

For situations where index expressions are used to access arrays, an if-statement check on the index expression may suffice. In case the index expression is controlled by a loop construct, the terminating value/condition should be changed to be within the size of the array, and the index variable should be checked for manipulations within the loop that can cause an out-of-bounds access.

In case the index variable controlled by a loop is used outside the loop (which is in itself questionable programming practice), one should be aware that the value of the index variable may be outside the range specified by the loop construct.

The two most common programming mistakes are (1) using the wrong inequality test on the loop conditional (generally,  $\leq$  when it should have been  $<$ ), and (2) using the final value of the loop index variable *after* the loop to index the array, when often the loop is written such that the final value is beyond the end of the array. The following two examples demonstrate each of these problems.

### EXAMPLE 1

```
#define TABLE_SIZE 20
int PowerOf2[TABLE_SIZE];

void initTable() {
    int j;

    PowerOf2[0] = 1;
    for (j = 1; j <= TABLE_SIZE; j++)
        PowerOf2[j] = PowerOf2[j-1] * 2;
}
```

In this example, the problem is that the conditional expression in the for-loop terminates when  $j > \text{TABLE\_SIZE}$ . During the last iteration of the loop,  $j == \text{TABLE\_SIZE}$ , which means that the assignment indexes one beyond the end of the array.

**EXAMPLE 2**

```

#define MAX_PATH
void MungeFilePath(char dos_path[])
{
    int i;
    char pathName[MAX_PATH + 1];    /* room for trailing '\0' */

    /* Convert MS-DOS path characters to UNIX form, add a
     * trailing '/' if necessary to prepare for later
     * concatenation with the file name.
     */

    for (i = 0; i < MAX_PATH && dos_path[i] != '\0'; i++) {
        if (dos_path[i] == '\\')
            pathName[i] = '/';
        else
            pathName[i] = dos_path[i];
    }
    if (pathName[i - 1] != '/')
        pathName[i++] = '/';
    pathName[i] = '\0';
    ...
}

```

This code fragment actually has two types of array bounds violations, both after the loop. The first defect occurs when the original path is `MAX_PATH` characters long and the last character stored in `pathName` is not a `'/'`. In this case, the logic is to append a `'/'` character, followed by a NUL character. Even though the programmer made the array one larger to hold the trailing NUL character, that is insufficient when both a `'/'` and a NUL character must be appended.

The second bug occurs when the initial string is zero length (i.e., the first character in `dos_path` is a NUL character). In this case, the first loop does not execute at all and the if-statement tests `pathName[i - 1]`. However, in this case `i` is zero, and this expression accesses a memory location before the beginning of the array!

**EXAMPLE 3**

```

int color[50];
...
for (i=0; i<50; i+=3) {
    color[i]= colorSet(i, ...);
    color[i+1]= colorSet(i, ...);
    color[i+2]= colorSet(i, ...);
}

```

An array declared as `a[n]` has `n` elements, indexed from 0 to `n-1`. In this example, `i` is incremented by 3 each time through the for-loop. During the last iteration through the loop, its value is 48. When `i=48`, `color[i+2]` accesses the out-of-bounds element `color[50]`.

## Uninitialized Variable

### DESCRIPTION

*Uninitialized variable* refers to a defect where local and dynamic variables are not explicitly initialized prior to use. Note that the ANSI standard requires that *global* and *static* variables are initialized (*ints* to 0, *floats* to 0.0 and pointers to `NULL`); therefore this defect is not reported for variables with either of these storage classes.

### IMPACT

Usage of uninitialized variables can cause unpredictable results in the program (because the value of the variable is essentially random), and in the worst case, a program exception.

### REPAIR

To repair this type of defect, the variable must be initialized to an appropriate value.

### EXAMPLE

```
int fun() {
    int len;
    if ((to - from) > 86400) {
        now = localtime(&from);
        len = strftime(&OdateToStr, 64, "%m/%d/%Y - ", now);
        now = localtime(&to);
        len = strftime(&OdateToStr[len], 64, "%m/%d/%Y", now);
    } else {
        now = localtime(&from);
        len = strftime(&OdateToStr[len], 64, "%m/%d/%Y", now);
    }
}
```

In the example above, the stack-allocated variable `len` is uninitialized when used in the `else` clause of the conditional. The `then` clause of the conditional is fine, but if the `else` path is taken, then the `len` used in the array access is an uninitialized variable. This may result in a program exception if the random value in `len` causes an out-of-bounds array read.





P.O. Box 478

Menlo Park, CA 94026-0478

+1 650-324-2510

[www.reasoning.com](http://www.reasoning.com)